Distributed Execution Engines

Abdul Saboor

Big Data Analytics - DIMA, Technical University Berlin,
Einsteinufer17, 10587 Berlin, Germany.
abdul.saboor@tu-berlin.de

1 Introduction

A distributed execution engine is a software systems which runs on a cluster of networked computers and presents the illusion of a single and reliable machine that provides large aggregate amount of computational and I/O performance. Each machine in these networked systems is capable of handling data-dependent control flow. The computational power effectiveness of a distributed execution engine is determined by the expressiveness of the data execution model which describes the distributed computations. Distributed executed engines are attractive because they shield developers from distributed and parallel computing's challenging aspects such as task scheduling, data synchronization, transferring data and dealing with failures in networks. A general purpose of high performance distributed execution engine for coarse-grain data parallel application is proposed which enables the developers to create easily large scale distributed application without requiring them to gain expertise on concurrency techniques beyond being able to draw a graph of the data dependencies of their algorithms. Based on the graph of DEEs, a job manager intelligently distributes the workload of network data processing so that the resources of execution engines are used efficiently. During the run time, a job manager automatically modifies the graph in order to improve the efficiency of data flow, and these modifications are based on run time information [1].

2 Distributed Execution Engines

The distributed execution engines such as Dryad, MapReduce, Pregel, CIEL, and Hadoop - provide the computational model that performs several independent sequential computations in parallel which comprises of subcomponents of a larger computation. However, for achieving the necessary data processing efficiencies, there are several requirements (or limitations) which are common in these models. One requirement is that every subcomponent must run in a separate isolated address space for its complete lifetime. Second requirement is that data exchanges can only occur between each round of processing (since communications are expensive). Generally, DEEs are used to process a large number of independent data items, and relatively these data items must be grouped into large batches (which is also referred to as partitions) with a comparative uniform execution time per batch to make the expensive cost of communication better and process creation or destruction. However, there are several requirements, that makes it difficult to solve the general optimization problems on a DDPE engine which typically require substantial communication between different tasks, particularly in the communication of intermediate results [2].

2.1 Distributed Data-Parallel Patterns and Execution Engines

The are many distributed data-parallel patterns which are identified recently and provide the opportunities to facilitate the data-intensive applications and their workflow, which can be executed with the split data in parallel on various distributed computing nodes. The main advantages of using these patterns are:

1. Support the distribution of data and parallel processing of data with distributed data on multiple nodes/cores.

2. Provide a higher-level programming model in order to facilitate the user program parallelization.

3. Follow a principle of ”moving computations to data” that reduces the data movements overheads.

4. Have a good scalability and efficiency in performance when executing on distributed resources .

5. Support run time features such as load balancing, fault-tolerance, etc.

6. Simplify the difficulties for parallel programming as compared to the traditional programming interfaces MPI [3] and openMP[4].

There are many execution engines which can be implemented over the distributed data-parallel patterns. MadReduce is such kind of example that has different DDP execution engine implementations. A most known MapReduce execution engine is Hadoop[5].

3 Main Characteristics of Distributed Execution Engines

The design purpose of Distributed Execution Engines is to consider the fundamental characteristics which are as follows;

- Task Scheduling

- Data Distribution

- Load Balancing
- Transparent Fault Tolerance
- Control Flow
- Tracking Dependencies

These are the fundamental aspects which different kinds of execution engines have some mechanisms according to their design in order to perform above certain tasks for completing their jobs.

4 Types of Distributed Exection Engines

There are different types of DEEs such as Dryad, Nephele, MapReduce, Hyracks, Pregel, CIEL, etc. and they used different types of systems, for instance, Dryad, Nephele, Hyracks, MapReduce are all use the Directed Acyclic Graph (DAG) system and in Pregel the Bulk Synchronous Parallel (BSP) system is implemented. The description of various execution engines is as follows:

5 Dryad Execution Engine - An Introduction

The Dryad distributed computing engine is designed by Microsoft to support the data-parallel computing and used in many important applications such as data-mining applications, image and stream processing and scientific computations. For these types of applications, distributing the data and computations on various clusters in an efficient way to process the high volume of data [6]. In addition, Dryad is a high-performance general-purpose distributed execution engine for running distributed applications on various cluster technologies. Dryad data-parallel applications combines the computational "vertices" with the communication "channels" to form a data flow graph. A Dryad runs the data-parallel applications on a set of available computers by executing the vertices of this graph, communicating through files, TCP pipes, and shared-memory FIFOs. Dryad also simplifies the task of implementing the distributed applications by addressing the issues in the preceding list, which are as follows:

- ◎ The Dryad execution engine handles the most difficult problems of large-scale distributed and concurrent applications such as delivering the data to appropriate locations, scheduling the use of computers and their CPUs, optimization, failure detection and recovering from computer failures or communication, and transporting data between vertices.
- ◎ The design of Dryad execution engine is to scale from powerful multi-core single computers through many small clusters of computer, to data centers with thousands of computers.

- ◎ Dryad scheduling vertices to run simultaneously on multiple computers or on multiple CPU cores within a computer.
- ◎ The application can discover the size and placement of data at runtime, and modify the graph as the computation progresses in order to make an efficient use of the available distributed network resources.
- ◎ Dryad has an expressive programming model which is designed to support the cluster-based computations. Dryad can also handle the large-scale data-parallel computations.

5.1 The Overview of Dryad System

A Dryad job is a mechanism for executing a distributed data-parallel application on a cluster and the structure of the Dryad job is determined by the flow of communication between vertices through various channels. A Dryad job systems is represented as Directed Acyclic Graph (DAG) where each vertex is a program and edges are denoted by channels, and there are many vertices in the graph while executing the cores in the computing cluster. Dryad has logical computation graph and during the runtime that is mapped automatically onto the available physical resource, and each channel at runtime is used to transfer a finite series of structured artifacts.

A schematic view of Dryad system organization is shown in the below Figure 1. A Dryad job is organized by the process which is called the "Job Manager" (represented in the figure as JM) that runs either within the cluster or on a user's workstation with the network access to the cluster. The Job Manager comprises the application-specific code for constructing the communication graph of job along with the library code in order to schedule the task across the available resources. All the data in Dryad system is sent directly to various available vertices and consequently the Job Manager is only responsible for controlling decisions and it does not obstruct the transfer for any kind of data.

The job manager (JM) communicate with name server (NS) to discover the list of available computers in the network. The job manager also maintains the job graph and schedules the running vertices (V) as computers become available while using the daemon (D) as a proxy. The vertices exchange the data through files, TCP pipes or shared-memory channels. The shaded portion in the figure is indicating the vertices in the job graph which are currently running. The cluster of name server (NS) that has list of available computer and it also expose the position of each available computer within the network topology so that task scheduling decision can be made according to their locations. A daemon (D) which in running on each network computer in the cluster and that is responsible for creating processes for the job manager. The first time vertex (V) is executed on a computer and its binary is sent from job manager to the daemon and then afterwards it is executed from the cache. The daemons that works as a proxy so that job manager can communicate with remote vertices and can monitor the state of computation that how much data has been read and written

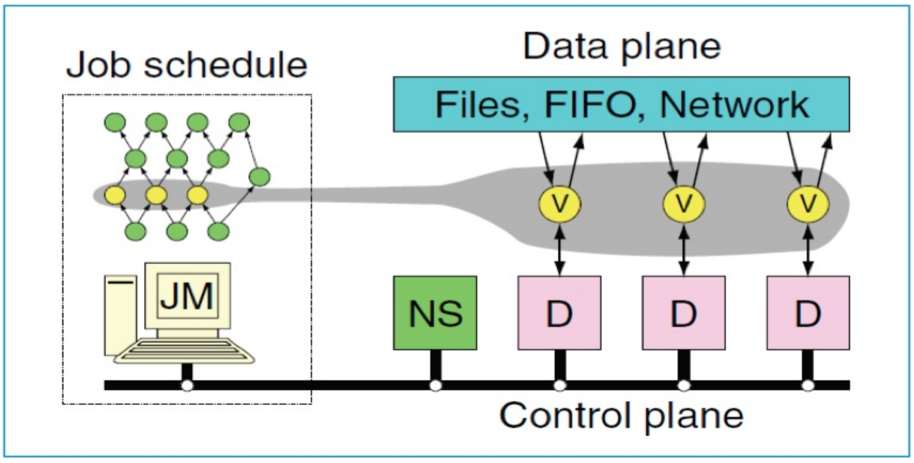

Figure 1: **Dryad system job organization [Ref. 7]**

on its channels. It is a simple to run name server and set a daemon on a client workstation in order to simulate a cluster and then run an entire job locally while debugging it [7].

5.2 Dryad Graph

A Dryad job consist of a chain of processes and with each process piping its output to the next process in the chain. Dryad extends the standard pipe model to manage the distributed applications that are running on a cluster. Dryad job execution plan is represented by a directed acyclic graph (DAG), which is also referred as Dryad graph.

The above figure has two main components:

1. The basic job execution plan - On the left of figure shows the structure of job operation as it runs on a single computer and it uses the standard pipe model, and with each stage of job piping the data move from one stage to next stage.

2. On the right side of Dryad graph, it is shown that Dryad might implement the operation as a distributed job running on a cluster of computers.

A Dryad graph is consist of some important features that are:

5.2.1 Input Data Partitions

A Dryad job starts with a collection of persistent input data, and it returns a processed version of that data to the application. The input data is typically partitioned into reasonable sizes which are distributed on various clusters before the job starts. The partitioned input data can be created and distributed in many ways, including:

1. By manual partitioning the input data into a set of files and copy each file to the client workstation.

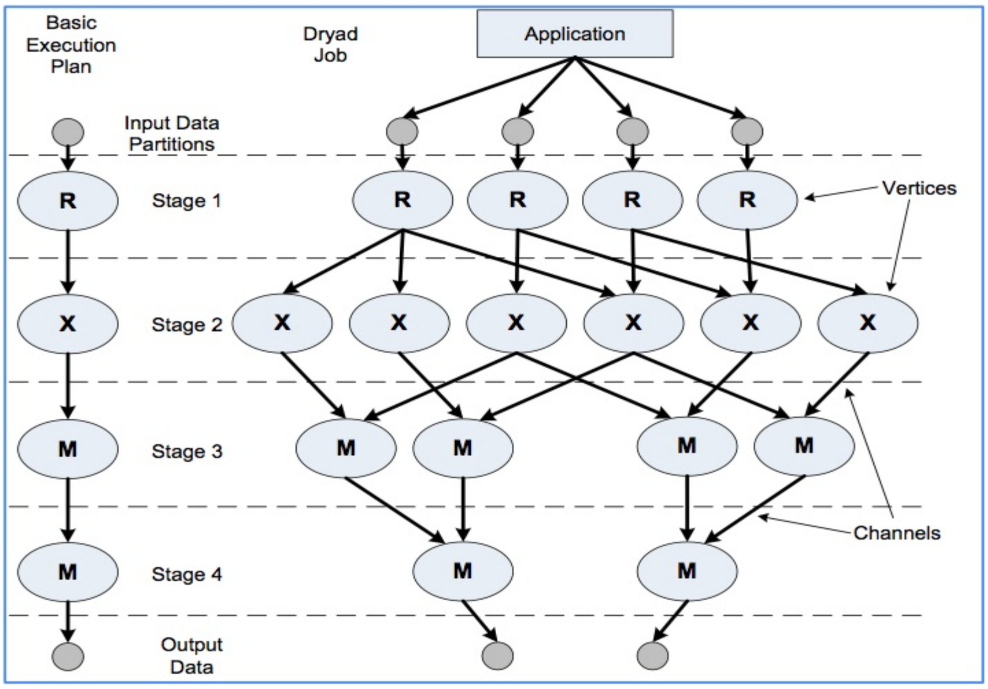

Figure 2: **Dryad Graph** [**Ref. 6**]

2. By using application to prepare the input partition and distribute the partitions across the cluster.

5.2.2 Stage

A Dryad job execution contains more than one stages and each stage is associated with an element of the Dryad basic execution plan.

5.2.3 Vertex

Each stage has more than one identical vertices and each vertex is an independent instance of data processing code for a stage, and it also processes a subset of that stage's data. The process of calculation uses mainly three vertex types that are labeled with R, M and X.

5.2.4 Channel

Vertices pass the processed data to the next stage of the Dryad job in the serialized form over the channels. A vertex can have a channel to more than one vertex in the next stage. Different types of channels ca be used at different point in the graph. These channels must connect the vertices so that the graph is acyclic. Channels are point to point and they connect the output of one vertex to the input of another vertex.

5.3 Failure Recovery and Job Monitoring

Dryad runs on cluster of computer hardware and it is common to get fail during execution on one or more client computers with or without warnings. The job manager monitors the overall status of all executing vertices and also analyze the transient or permanent failures. In Dryad system, a node must report within a set timeout value and if a node fails to report within that particular period of time, the job manager assumes that the vertex has been failed or there is any other problem such as computer crashed, and it initiates the recovery procedures for failure node.

6 MapReduce - The concept

MapReduce is a programming model and associated implementation for processing and generating large scale data sets. Users need to specify a map function that processes a record (key/value pair) to generate a set of intermediate record (key/value pair), and a reduce function that merges all intermediate values associated with the same intermediate key. The programs that are written in this type of functional style are automatically parallelized and executed on the large cluster of commodity machines. In MapReduce, the run-time system takes care of details of partitioning the input data, scheduling the program for execution across a set of machines, handling the failures of machines, and managing the required inter-machine communication. This allows the programmers to easily utilize the resources of a large scale distributed system without having any kind of experience. The implementation of MapReduce runs on a large cluster of commodity machines and that is highly scalable such as a typical MapReduce computation processes of many terabytes of data on thousands of computers. The system is easy to use for programmers as hundred of MapReduce programs have been implemented and they upward one thousand or more of MapReduce jobs are executed on Google's clusters on daily basis.

6.1 MapReduce System Overview

The Map instances are distributed across multiple network computers by automatically partitioning the input data into a set of M splits. The input files splits can be processed in parallel on various computers. The Reduce instances are also distributed by partitioning the intermediate key space into R pieces using a partitioning function(e.g. hask(key) **mod** R). The number of partitions (R) and partitioning functions are specified by the users. Figure 3 shows the overall flow of a MapReduce operation in the implementation. When a user calls the MapReduce function, then the following sequence of actions occurs which are:

1. The MapReduce for the first splits in the user program the input files into M pieces of typically 16 to 64 megabytes (MB) per piece (i.e. controllable by the user via an optional parameter). Then It starts up to many copies of the program on a cluster of machines.

2. One set of copies for the Master program is special and the rest copies are sent to workers that are assigned by the master. There are M map tasks and R reduce tasks to assign. The master picks up the idle workers and assign each of one a map task or a reduce task.

3. A map task that is assigned to a worker who reads the contents of the corresponding input split. It parses out the key/value pairs of input data and passes to each pair to the user-defined *Map* function. The intermediate key/value pairs are buffered in memory that produced by the *Map* function.

4. The buffered pairs are written to the local disk periodically and partitioned into R region by using the partitioning function. The locations of these buffered paris on the local disk are passed back to the master who is responsible for forwarding these locations to the reduce workers.

5. When a master notify the a reduce worker about these locations, it uses remote calls procedure to read the buffered data from the local disks of the map workers. When a reduce worker has read all the intermediate data, it sorts the data by intermediate keys so that all occurrences of the same key are grouped together. The sorting of data is needed because many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory then an external sort is used.

6. The reduce worker iterates this process over the sorted intermediate data and for each unique intermediate key encountered and it passes the key and also the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partitioned.

7. When all the map tasks and reduce tasks have been completed successfully, then the master awake the user program. At this point , the *MapReduce* call in the user program returns back to the user code.

After successful completion of tasks, the output of MapReduce execution is available in the R output files (one per reduce task with the file names as specified by the user). Users don't need to merge these R output files into one file because they often pass these files as input to another MapReduce call or use them from another distributed application which is able to deal with input that is partitioned into multiple files [8]. The figure 4 gives more details about the MapReduce execution process that is given below:

6.2 Data Structure of The Master

The master keeps several data structures, for each map task and reduce task its stores the various states such as idle, in-progress/in-process or completed, and the master also keep the identity of the worker machine (for non-idle tasks). The master is the main channel through which the location of intermediate file regions is disseminated

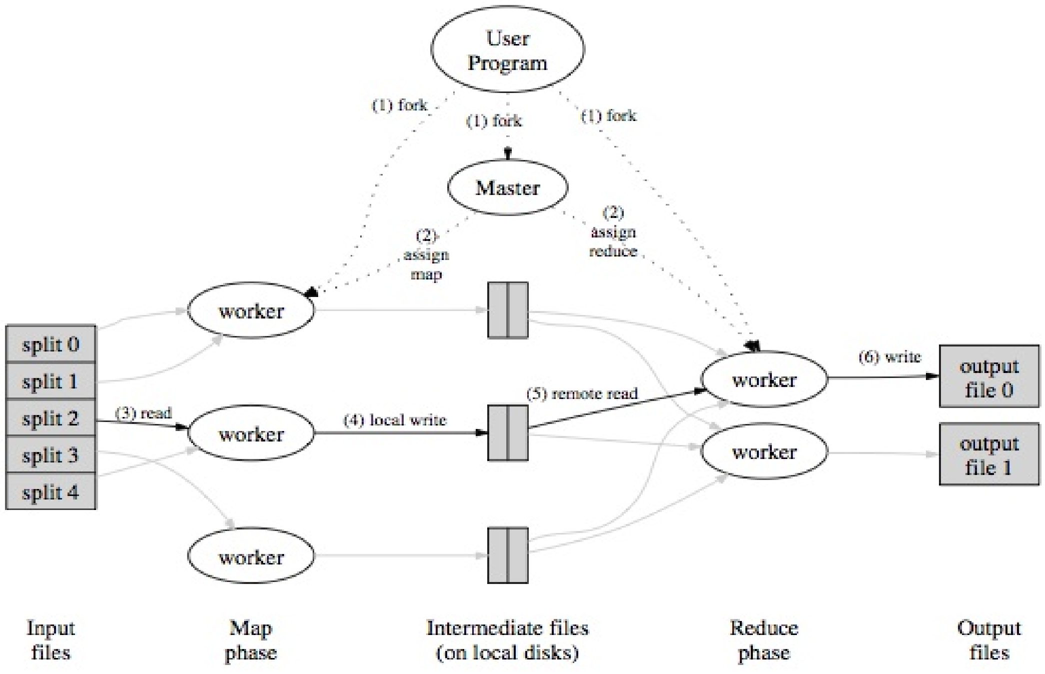

Figure 3: **MapReduce Execution Overview [Ref. 8]**

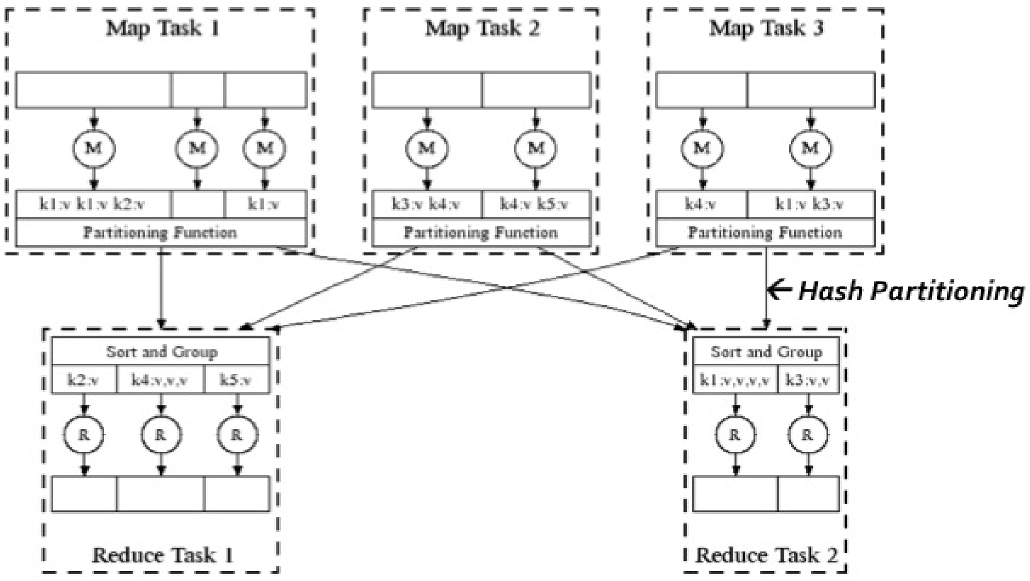

Figure 4: **MapReduce Execution Overview**[**Ref. 14**]

from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions that are produced by the map task. As the map tasks are completed then the updates to this location and size information are received. The information is thrusted to worker with the increment that have in-progress reduce tasks.

6.3 Task Granularity

The map phase is subdivided into M pieces and the reduce phase into R pieces as it explained in above figure 4. Considerably, M and R should be large than the number of worker machines. Each worker machine performs many different task in order to improve the dynamic load balancing as well as speeds up to make recovery when a worker gets fail.

6.4 Fault Tolerance

The designed purpose of the MapReduce library is to help the process of very large amounts of data using hundreds or thousands of machines, the library must deal with machine failures very effectively. The failures could be:

6.4.1 Worker Failure

The master pings each worker periodically and if there is no response comes from worker in a certain amount of time then the master marks that worker as failed. Any map task which is completed by worker are reset back to their initial idle state, and it becomes eligible for scheduling on other workers. Similarly, if there is any map task or reduce task is in-progress on a failed worker that is also reset to idle state and becomes eligible for rescheduling. The completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machines and that is inaccessible. Completed reduce tasks do not need to re-executed because their output is stored in a global file system.

MapReduce is resilient for the large-scale worker failures. For example, during the one MapReduce operation the network maintenance on a running cluster was causing a group of almost 80 machines at a time to become unreachable for several minutes. The master of MapReduce simple re-executed the work done by the unreachable workers, and continued to make further progress, eventually the completion of MapReduce operation.

6.4.2 Master Failure

The master writes periodic checkpoints of the master data structure as it is described above. If the master task gets fail, then a new copy can be started from the last checkpoint state. However, there is only a single master and its failure is unlikely.

Therefore the current implementation abort the MapReduce computation if the master gets fail. Clients can consider this type of condition and can retry the MapReduce operation if they like to do it again.

7 Hyracks - An Introduction

Hyrack is a new software platform which is flexible, extensible, partition-parallel framework designed to run efficient data-intensive computations on large shared-nothing clusters of commodity computers. Hyrack computational model is represented as DAG (Directed Acyclic Graph) of data operators and connectors. Operators run on partitions of input data and produce partitions of output data while connectors repartitions the operators' outputs to make the newly produced partitions available at the consuming operators. Hyracks has the significant promise as a next-generation platform for data-intensive applications.

There are some important which are as follows:

1. Hyracks has a built-in collection of operators that can be used to assemble data processing jobs without needing to write processing logic similar to Map and Reduce code.

2. The rich API enables the Hyracks operator implementations to describe the operators' behavioral and resource usage characteristics to the framework for better jobs' planning and runtime scheduling that utilize their operators.

3. The inclusion of a Hadoop compatibility layer which enables the users to run existing Hadoop MapReduce jobs unchanged on Hyracks as an Initial strategy "get acquainted" as well as a migration strategy for "legacy" data-intensive applications.

4. The initial method for Hyrack tasks scheduling on a cluster that involves the basic fault recovery, to guarantee the job completion through restarts. The performance overview of one class of job on Hyracks and Hadoop under various failure rates to express the potential gains offered by a less pessimistic approach to fault handling.

5. The Hyracks as an open source platform can be utilized by others in the data-intensive computing community.

7.1 Hyracks System Overview

As Hyracks is a partitioned-parallel data flow execution platform that runs on shared-nothing clusters of commodity computers. The large collection of data items are stored on local partitions and distributed across the various nodes of the cluster. A Hyracks job is a unit of work that is submitted by a client and processes one or more collection of data in order to produce one or more output collections in the form of partitions. Hyracks infrastructure and programming model efficiently divide computations on large data collection (spanning on multiple network machines) into computations that work on each partition of the data separately. For example, a Hyracks job is a dataflow DAG which is composed of operators (nodes) and connectors (edges). Operators represent the job's partition-parallel computation steps, and connectors represent the distribution of data from one step to another step. Internally, an individual operator consists of one or more activities (internal sub-phases or sub-steps). At the runtime, each operator is considered as a set of tasks (identical) that are cells of the activity and that operate on individual partition of the data flowing through the activity. Hyracks system architecture is showed in the figure 5.

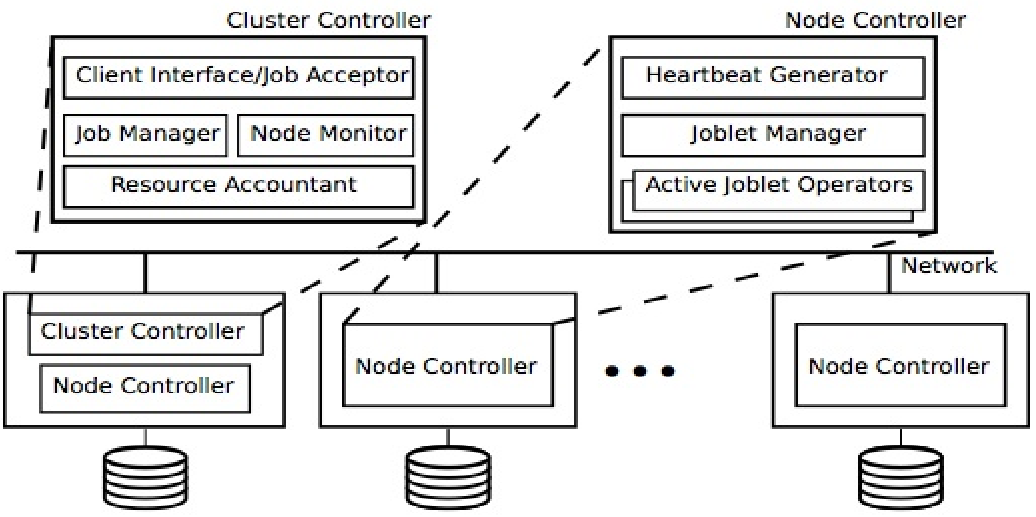

Figure 5: **Hyracks System Architecture [Ref. 9]**

7.2 Hyracks Data Treatment

In Hyracks, the data flows in the form of records between operators and connectors that can have an arbitrary number of fields. Hyracks provides support for demonstrating data-type specific operations such as hash functions and comparison. Each type of field is described by providing an implementation of a descriptor interface which allows Hyracks to perform serialization and deserialization. For example, the basic types such as number and text, Hyracks library consists of pre-existing type descriptors. Hyracks uses a record as a carrier of data that is generalization of the (key, value pair) concept that is used in the MapReduce and Hadoop as well. The advantage of this generalization is that operators do not have to artificially package or re-package the multiple data objects into a single "key" or "value" object. The use of multiple fields for hashing or sorting becomes natural in the Hyracks model because Hyracks operator library includes an external sort operator descriptor that can be parameterized with the fields in order to use them for sorting along with comparison function. The Type descriptors in Hyrack is similar to Hadoop but the important difference is that Hyracks does not need the object instances that flow between operators to themselves and implement a specific interface. This make able the Hyracks to directly process the data that is produced and/or consumed by the systems without having knowledge about Hyracks system and its interface [9].

7.3 Hyracks MapReduce user Model

In MapReduce and Hadoop, data is initially partitioned across the various nodes of a cluster and stored in a distributed file systems (DFS). Data is also represented as (key or value pairs) and the desired computation is defined by using the two functions:

map (k1, v1) $\longrightarrow$ list(k2, v2);
reduce (k2, list(v2)) $\longrightarrow$ list(k3, v3).

A MapReduce computation starts with a map phase in which the Map functions are applied in parallel on different partitions of input data. The key/value pairs output by each map functions are then hash-partitioned on their key. For each partition, the paris are sorted by their key and sent accross the cluster in shuffle phase. At each receiving node, all of the received partitioned are merged in sorted order by their key. All of the pair values sharing a given key are then passed to a single reduce call. The output of each reduce function is finally written to a distributed file in the DFS. As it is described in the Figure 6 below:

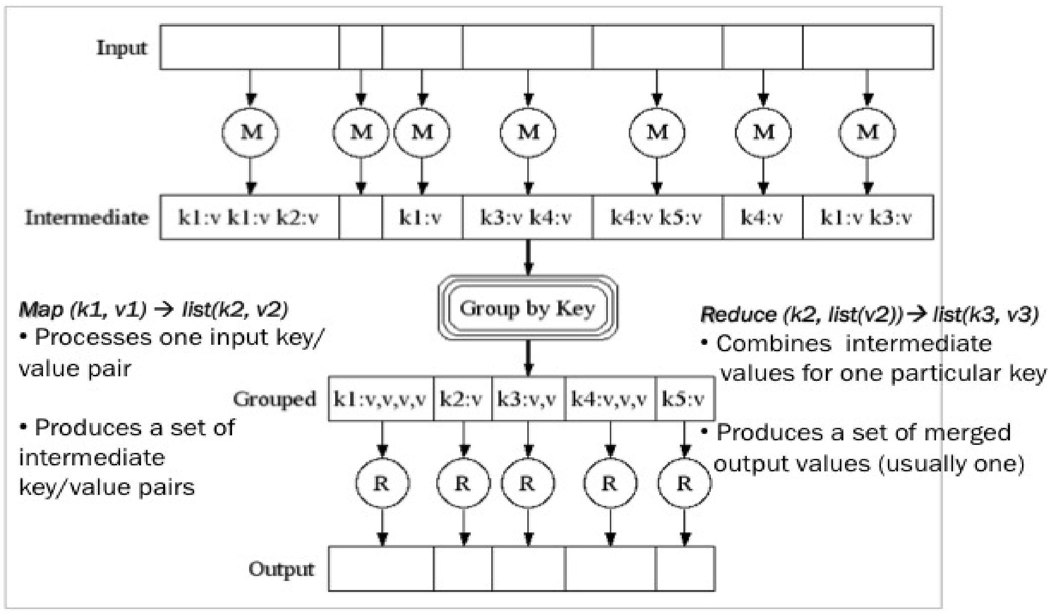
Input
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Map (k1, v1) → list(k2, v2)
• Processes one input key/value pair
• Produces a set of intermediate key/value pairs
Reduce (k2, list(v2)) → list(k3, v3)
• Combines intermediate values for one particular key
• Produces a set of merged output values (usually one)
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
Output

Figure 6: **Hyracks System Architecture [Ref. 14]**

7.4 Hyracks Task Execution Layer

Once a stage of a Hyracks job becomes ready to run then cluster controller (CC) activates the stage on the set of Node Controllers (NC) that have been chosen to run the stage and then waits until the stage completes or an NC failure is detected. There are two important phases which are as follows:

1. **Task Activation:** When a stage becomes ready to run then Cluster Controller sends messages to the Node Controller for participating in that stage's execution to perform task activation in three main steps:

    a. **Receiver Activation:** The Cluster Controller initiates a Receiver activation request to every Node Controller that participates in the stage's execution. Each Node Controller on the reception of this request, creates the Task objects which have been designated to run at Node Controller (determined by scheduler). For each Task which accept the input from other Tasks, then it creates an endpoint that has a unique network address. The mapping from a Task object's input to the endpoint address is transmitted back to the Cluster Controller as response to this request.

    b. **Sender-Receiver Pairing:** Once the Cluster Controller receives the Receiver Activation responses consisting local endpoint address from the Node Controllers, then it combines all together in order to create a global endpoint address map. This global map is broadcast to each Node Controller so that the sender side of each Task object at that Node Controller knows the network addresses of its consumer Tasks.

    c. **Connection Initiation:** Once pairing has been completed on all Node Controllers then Cluster Controller informs all of the Node Controllers that the Tasks can be started.

2. **Task Execution and Data Movement:** The unit of data which is consumed and produced by Hyracks tasks is called a Frame. A frame is a fixed-size configurable chunk of contiguous bytes. A producer of data packs a frame with a sequence of records in a serialized format and sends it to the consumer who then interprets the records. Frames never contain partial record or a record never split across frames. An ordinary approach to implement tasks that process records is to deserialize then into Java objects and then process them. In this way there are many objects can be created, resulting in more data copies for creating the object representation and a burden comes on garbage collector (later on) for reclaiming the space for those objects. To handle this kind of problem Hyracks provides interfaces for comparing and hashing fields that can be implemented to work off the binary data in a frame. Hyracks implementations includes these interfaces for the java types such as integer, float, string, etc. Hyracks provides the Task implementor with the helper functions to perform basic tasks on the binary data such as to copy the specific fields, copy the complete records and concatenate the records from source frame to target

frames.

The runtime logic of a task is implemented as a push-based iteration which receives a frame at a time from its inputs and pushes the result frames to its consumers. The task runtime only implement the logic to process the stream of the frames which belongs to one partition without relating to any repartition of inputs or outputs which may require and any repartitioning can be achieved by Connectors. At runtime the connector has two sides, the send-side and receive-side. The send-side instances are co-located with the associated producer tasks, and receive-see instances are co-located with the corresponding consuming tasks. When a connector of a send-side instance receives a from its producing task, it applies the redistribution logic to move records to the relevant receive-side instances of the connector. Hyracks redistributes records based on their hash-value because of hash-partitioning connector. The send-side instance of the connector inspect each record in the received frame and compute the hash value for it and also copy it to the target frame indicating the appropriate receive-side instance. When the target frame is full, then the send-side instance sends the frame to the receive-side instance. If the send-side and receive-side instances of a connector are on different worker machines then sending a frame does write it out to the network layer. The network layer in Hyracks is responsible for ensuring that the frame gets to its destination[10].

7.5 Hyracks Fault Tolerance

A Hyracks system is particularly designed to run on a large number of commodity computers and it must be capable for detecting and reacting against possible faults that might occur during its regular operations. Hyracks is in better position to use operator and connector properties to achieve the same degree of fault tolerance while doing less work along the way. Hyracks job is less likely to hit by a failure during its execution in first place.

8 Pregel Execution Engine

There are many practical problems that mainly concern with large graph. The large scale of these graphs (billions of vertices and edges) poses challenges for their efficient processing. In this case, there is a need for a computational model for this task and that is Pregel computation model. In Pregel, programs are expressed as a sequence of iteration, in which each of a vertex receives messages sent by the previous iteration, send messages to other vertices and modify its own state that of its outgoing edges. The vertex-centric approach is flexible enough to express a broad set of algorithms. The design purpose of this model is efficient processing, scalable and fault-tolerant implementation on clusters of thousands of commodity computers, and its synchronicity that makes reasoning about program easier. Distribution related details are hidden in system and kept behind an abstract API. The result is a framework for processing large-scale graphs which is expressive and easy to program.

The high-level organization of Pregel programs are represented by Bulk Synchronous Parallel model [11]. Pregel computations contains a sequence of iterations that are called *supersteps*. During a *superstep* the framework refers to a user-defined function for each vertex, conceptually in parallel processing. The function specifies the behavior of a single vertex V and a single superstep S. It can read messages that sent to V in superstep $S - 1$, it send messages to other vertices that will be received at superstep $S + 1$, and modify the state of V and its outgoing edges. Messages are sent along outgoing edges but a message can be sent to any vertex whose identifier is unknown. The synchronicity of Pregel model makes it easier to reason about program semantic when implementing algorithms, and it ensures that Pregel programs are inherently free of deadlocks and data races are common in asynchronous systems. The performance of the Pregel programs should be competitive with that of asynchronous systems given enough parallel slack [12, 13] because typical graph have much more vertices than machines. One should be able to balance the machine loads so that the the synchronization between supersteps does not add excessive latency rate.

8.1 The Computation Model

The input to a Pregel computation model is a directed graph in which each vertex is uniquely identified by a string vertex *identifier*. Each vertex is associated with a modifier, user defined value. The directed edges are associated with their source vertices and each edge contains a modifier, user defined value and a target vertex identifier. A typical Pregel computation model consists of input, when the graph is initialized, that is followed by a sequence of supersteps separated by global synchronization points until the algorithm terminates and finishes with output. Within each superstep the vertices are computed in parallel, and each vertex is executing the same user-defined function that describes the logic of given algorithm. Algorithm termination based on each vertex voting to halt. In the superstep 0, every vertex is in the active state and all active vertices participate in the computation of any given superstep. A vertex deactives itself by voting to halt which mean that it has no further work. The Pregel framework will not execute that vertex unless it receives a message. If vertex reactivated by a message then a vertex must explicitly deactivate itself again. The whole algorithm terminates when all the vertices are inactive at the same time and there are no jobs in transit. As it is showed in the Figure 7 below:

8.2 Pregel System Architecture

The Pregel library divides a graph into partitions, each partition consisting of a set of vertices and those vertices' outgoing edges. The task of each vertex to a partition depends only on the vertex ID, and it is possible to know which partition of a given vertex belongs to even that vertex is owned by a different machine or even the vertex does not exist yet. The default partitioning function is just hash(ID) function mod N,

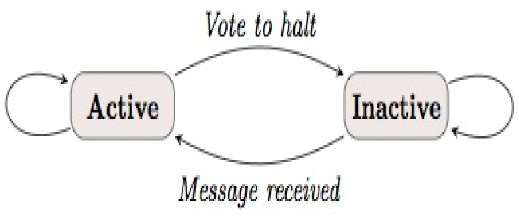

Figure 7: **Pregel Computation Model [Ref. 13]**

where N represents the number of partitions but users can replace it. The execution of Pregel program contains several steps which are as follows [14]:

1. Multiple copies of user program start execution on a cluster of machines and these copies act as a master. It is not assigned to any portion of graph but it is responsible for coordinating worker activity. The workers discover the master's location through cluster management system's name service, and send registration messages to the master.

2. The master discovers how many partitions the graph will have and assigns each worker machine one or more partitions. Having more than one partitions per worker establish the parallelism among the partitions and better load balancing, and will improve the performance.

3. The master assigns a portion of the user's input to each worker and that input treated as a set of records, each of which consists of an arbitrary number of vertices and edges.

4. The master gives instruction to each worker to perform superstep. The worker loops through its active vertices while using one thread for each partition. The worker calls computer( ) function for each active vertex, for delivering the messages that were sent in the previous superstep. Messages are sent asynchronously to enable overlapping of computation, communication and batching but are delivered before the end of the superstep. When worker finish the job it responds to the master and tells that how many vertices will be active in the next superstep.

5. After the computation process halts, the master may give the instructions to each worker for saving its portion of the graph.

8.3 Fault Tolerance in Pregel

In Pregel, fault tolerance is achieved through checkpointing. At the beginning of each superstep, the master gives instruction to workers to save that state of their partitions for persistent storage including vertex values, edges values and incoming messages, and the master separately saves the aggregator values. The failures of worker are detected by using a regular ”ping” messages that the master issues to different workers. If a worker does not receive a ping message after a specific time period, the worker process terminates. And if the master does not get any response back from a worker then master marks that worker process as failed. When one or more workers get failed, then the current state of the partitions that assigned to these workers is lost. The master reassigns the graph partitions to currently available set of workers and they all reload their partition state from the most recent available checkpoint at the beginning of a superstep *S*. The checkpoint could be the several supersteps earlier than the latest *S'* that is completed by any partition before the failure.

9 Conclusion

There are number of Distributed Data Parallel Execution Engines, some of them employed similar data patterns but some of them have different capabilities and characteristics in terms of task scheduling, data distribution, controlling the flow of data, fault-tolerance and load balancing for better performance, etc. The implementations directed to methods for a distributed data-parallel execution system (which comprises an engine and a director which generates distributed jobs for the engine, it starts from programs or queries and/or front end library that generates queries for the director in the process of solving an optimization program) to split a computational problem into a plurality of sub-problems using a branch-and-bound algorithm, designating a synchronous stoppage time for a ”plurality of processors” (e.g. a cluster) for every round of execution. Processing the search tree recursively using a branch-and-bound algorithm in multiple rounds for determining if there is a need of further processing based on the processing round state data, and then terminating the plurality of processors when processing is completed. Several implementations are directed to a library which enables massively parallelism and for solving hard problems using distributed execution of optimization algorithm by performing a complete search of the solution space using branch-and-bound algorithm (i.e. by recursively splitting the original problem into many simpler sub-problems). For several implementation, parallelism and distribution can be handled automatically and can be invisible to the user, and several implementations may be implemented on the top of distributed data-parallel execution engines, for example, Map-Reduce, Hadoop, DryadLINQ, despite the use of a constrained application model (with restricted communication patterns), that implementations can scale linearly in performance perspectives with the number of machines without much overhead [15].

References

[1] Derek Gordon Murray. A distributed execution engines supporting data-dependent control flow. PhD Thesis, University of Cambridge, Jully 2011, pp. 15.

[2] Daniel Deling, Mihai Budiu, Renato F. Werneck. Branch-And-Bound Distributed data-parallel execution engines. *http://www.google.com/patents/US20120254597.* Microsoft Corporation, CA, US. October 2012.

[3] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd Edition, Scientific and Engineering Computation Series, MIT Press, Cambridge, MA, USA, 1999, pp. 51-59.

[4] B. Chapman, G. Jost, R. van der Pas, D. Kuck, Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press, Cambridge, MA, USA, 2007, pp. 23-30.

[5] Dianwu Wang, Daniel Crawl, Ilkay Altintas, A Framework for distributed data-parallel execution in the Kepler scientific workflow system. *In the Proceedings of the International Conference on Computational Science* San Diego Supercomputer Center CA, USA: Elsevier, April 2012, pp. 75-83.

[6] Microsoft Research, Dryad and DryadLINQ: An Introduction, *http://research.microsoft.com/en-us/collaboration/tools/dryad.aspx,* Version 1.0.1, November 2010.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: Distributed data-parallel programs from sequential building blocks. *In the proceeding of European Conference on Computer Systems (EuroSys)*, Lisboa, Portuagal, March 2007.

[8] J. Dean, Sanjay Ghemawat, MapReduce: Simplified data processing on large clusters. *In the proceeding of 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, USA, December 2004, pp. 137-150.

[9] V. Borkar, M. Carey, R. Grover, N. Onose, R. Vernice, Hyracks: A flexible and extensible foundation for data-intensive computing. *In the proceeding of International Conference on Data Engineering (ICDE)*, Hannover, Germany, April 2011.

[10] Leslie G. Valiant, A Bridging model for parallel computation. Magazine: Communication of ACM, August 1990, Vol. 33, Issue No. 8, pp. 103-111.

[11] Paris C. Kanellakis, Alex A. Shvartsman, Book: Fault-Tolerant Parallel Computation, ISBN: 0792399226, Kluwer Academic Publisher, 1997.

[12] Grzegorz Malewicz, A Work-Optimzal deterministic algorithm for the Certified Write-All problem with non-trivial number of Asynchronous Processors. SIAM Journal on Computer, 2005, Vol. 34, Issue No. 4. pp. 993-1024.

[13] G. Malewicz, M.H. Austern, A.J.C Bik, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing. *In the proceeding of the 2010 ACM SIGMOD International Conference on Management of data.* ISBN: 978-1-4503-0032-2, Indiana, USA, 2010, ACM, pp. 135-146.

[14] V. Borkar, M. Carey, R. Grover, N. Onose, R. Vernice, Hyracks: A flexible and extensible foundation for data-intensive computing. *In the proceeding of International Conference on Data Engineering (ICDE)*, Hannover, Germany, April 2011. University of California, Irvine, Presentation.

[15] Daniel Deling, Mihai Budiu, Renato F. Werneck. Branch-And-Bound Distributed data-parallel execution engines. *http://http://www.faqs.org/patents/app/20120254597*. Microsoft Corporation, CA, US. October 2012.