Can MapReduce be Implemented with MPI? A Comparison and Implementation Exploration of Two Well-Known Parallel Programming Frameworks

Nitya Dhanushkodi

Abstract

This paper aims to understand two parallel programming paradigms, Message Passing Interface (MPI), and MapReduce. It will detail the MapReduce algorithm, how it is implemented in products like Hadoop, and how its performance could be improved with an efficient and robust implementation using MPI. It will present an overview of the differences between MPI and MapReduce, most importantly how MPI is a general framework which works well for task parallelism and how MapReduce is a specific implementation of a parallel algorithm, that lends itself elegantly to a few problems involving data parallelism. Since MapReduce is simply a more specific framing for a problem, it can, in fact, be implemented using MPI. This paper also talks about the improvements and limitations of implementing MapReduce in MPI versus the implementation in Hadoop. MapReduce could be more speed efficient if implemented using MPI, but fault tolerant characteristics important to the MapReduce paradigm would be hard to replicate with MPI.

Index terms— MPI (Message Passing Interface), MapReduce, Hadoop, parallel computing paradigms, implementation

I. Introduction

Parallel computing is implemented in so many different ways, including the same programs running on different machines with different data to compute some combined value (SPMD), on multiple threads, across distributed memory with message passing, or with data parallelism. This paper dives into two well-known frameworks for parallel computing: the Message Passing Interface (MPI) and MapReduce. The important distinctions to make in such parallel frameworks are in the following points:

- Shared or distributed memory
- Centralized or distributed coordination

These points will be discussed in relation to MPI and MapReduce, to explain what the two frameworks do, and how they are similar and different to each other and to other forms of parallel computing.

Message Passing Interface (MPI) is a standard for passing messages between programs running on different nodes, with distributed (not shared) memory. MPI itself is a standard, and there are several popular implementations being used, such as OpenMP and MPICH2. MPI, being more general, can use either centralized or distributed coordination. When writing programs with MPI, the user decides exactly how each process communicates with any other process. For instance, the user has the granularity to decide whether process number 3 will send a message to process number 7. Or the user may decide that process number 0 will serve as the master process that directs most communications. This layer of granularity is abstracted away in MapReduce. [1]

MapReduce is a framework used to easily express certain parallel programming algorithms by simply implementing two functions: map and reduce. It abstracts away the communication, and technically, the user does not even have to know how their data is being synchronized. All they have to do is write what mapping function will be applied to the input data in parallel, and then how to compute some combined value from that mapping. In order to better understand these differences, we will look at MPI and MapReduce in a little more detail.

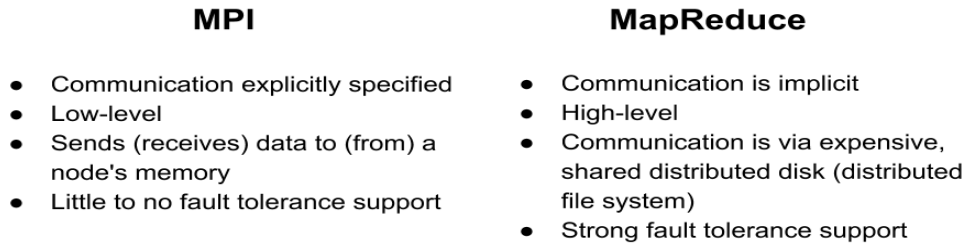


Fig. 1: This figure lists a few simple differences between MPI and MapReduce. It gives an idea of the strengths and weaknesses of both paradigms[3].

A. Message Passing Interface (MPI) Overview

MPI is a message passing standard, used for writing programs that do not share the same memory and need to pass messages between processes. In an MPI program, the user is writing the code that will be on multiple machines. Each process running the code has a unique process id, and there are often conditionals in the code determining what a process should do, given its own particular id. For instance, if the program had a centralized "master" process, the master process code would be in separate conditionals from a "worker" process's code. MPI also has the flexibility to have all processes not follow the "master-worker" pattern, with completely decentralized coordination. In an MPI program, the user must initialize MPI, find out the rank of this process's id, and execute the appropriate code. The user must understand what every individual process is doing. For instance, in MapReduce, the user knows that every worker process will be executing the map function, and the communication between processes before the reduce function is executed is already implemented by the framework. However, for MPI, the user must know exactly what code is being executed by each process, and whether that process is getting the information it needs. [1]

B. MapReduce Overview

The MapReduce framework allows the user to define two functions. The first is `map` function, which takes in an input vector of N key-value pairs (k, v). Each key k is in the set of keys K_i, and each value v is in the set of values V_i. The output of the map function are intermediate key-value pairs (g, w) where $g \epsilon K_r$ and $w \epsilon V_r$. The second function is `reduce`, which takes in a key g where $g \epsilon K$, and a list of values v' that are associated with key g, and generates an output return value computed from its input(s).

In implementations of this framework, there are several processes (which can be scaled) that run the mapping function on different parts of the data until all of the data has been mapped. Then, the framework does a grouping step to get the inputs ready for the reduce function, and the reduce function is run on many processors for all of the keys g in K_r to obtain a final result[5].

In the framework, `map` is called for each of the N key-value pairs (k, v), and emits output key-value pairs (g, w), and the framework collects these intermediate pairs. Then, for every key g in K_r, the framework finds all of that keys associated values, possibly emitted by even other mapping processes or the one running on this machine. The framework has the key g and vector v' of the values associated with that key to pass to the reduce function, which uses these inputs to return a single output. Figure 2 shows how the framwork must communicate when running these functions on different processors[5].

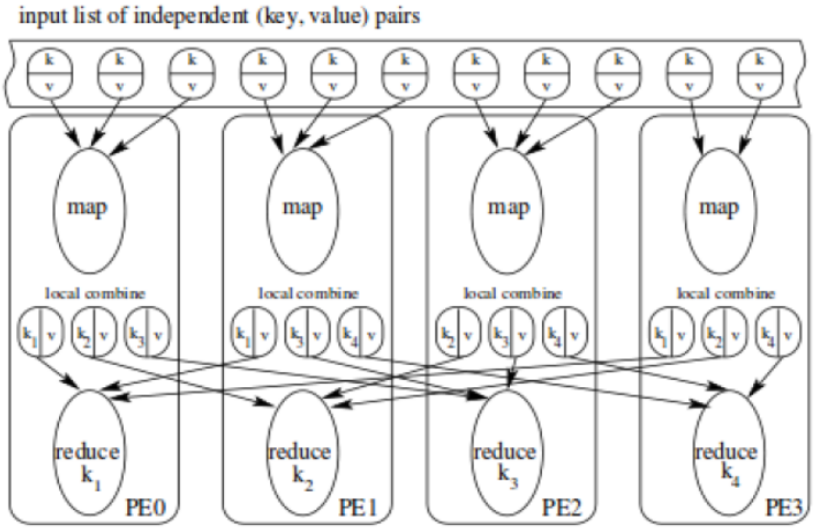


Fig. 2: The communication between processors in the MapReduce Framework. Each PE_i is a processor in this diagram[5].

The processors need to communicate three important pieces of data:

1) The N input key-value pairs (k, v)
2) The emitted intermediate pairs (g, w) need to be grouped by key g and have g and the list of values associated with g sent to the reduce processes
3) The reduce processes need to output the result

All of the input data in a MapReduce implementation is stored on a Distributed File System (DFS). All of the processors know where to look for their partition of data for the mapping step.

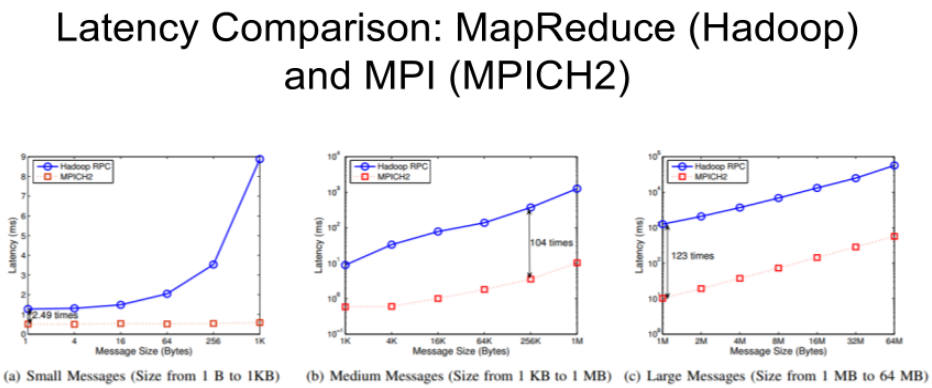
II. Comparing MPI and MapReduce

The general differences between MPI and MapReduce are shown in Figure 1. Those differences are based on the paradigms themselves. In comparing MapReduce application implementations to MPI implementations, we will be able to see that MPI can make the communication in MapReduce faster, making MapReduce more efficient[6]. This is described in the latency discussion. However there are certain applications MapReduce works better for, and certain applications that MPI works better for, which is described in the applications section.

A. Latency

Comparing the performance of communication in the Hadoop MapReduce implementation to the MPICH2 MPI implementation will show whether MapReduce's communication time can be reduced significantly implementing it using network communication as used in MPI implementations. [6]

Lu et al. performed an experiment with a simple relay communication in both of these implementations to compare the latencies for communication. In MPICH2, they use the common `MPI_Send` and `MPI_Recv` functions. `MPI_Send` and `MPI_Recv` are both point to point communication functions, meaning that one process is sending messages over the network to another process. There are other functions, such as `MPI_Barrier`, `MPI_Broadcast`, and `MPI_Reduce` called collective operations which will be discussed later. For Hadoop, Lu et al. use the Hadoop RPC communication mechanism. Figure 7 shows the communication latencies comparison between these two point to point communications.



(a) Small Messages (Size from 1 B to 1KB) (b) Medium Messages (Size from 1 KB to 1 MB) (c) Large Messages (Size from 1 MB to 64 MB)

Fig. 3: Comparing communication latencies with messages of different sizes in MPICH2 and Hadoop RPC[6].

For the Hadoop RPC test, they use a basic class extending from the `VersionedProtocol` interface with a simple receiving method, which checks the received data size, and sends the data back to the invoker. For the MPICH2 test, they use `MPI_Send` and `MPI_Recv` with data size as the message as well.

From Figure 7 it can be seen that for all 3 message sizes, the MPICH2 communication is faster than Hadoop RPC. So, it is likely that a MapReduce implementation could benefit in terms of communication efficiency if the point to point communication was implemented in MPI. These communications would mostly occur during the shuffling phase of the MapReduce paradigm.

B. Applications

Applications can show differences in planning algorithms when implementing parallel code in MapReduce or MPI. Here, matrix multiplication will be described in both of these paradigms. Then, a very simple example for counting words will be shown in the MapReduce paradigm to illustrate the ease of use for MapReduce when problems can be elegantly solved using it.

1) Matrix Multiplication with MPI: When computing a matrix multiplication A x B = C, then cell (i, j) of C needs row i from A and col j from B. The algorithm below uses a master-worker pattern with MPI to compute C.

1) `Partition: Each worker computes a row or many rows of C. The master must divide these tasks such that each worker is computing an equal number of rows in C. Worker n computes C starting at (n-1)*numRows/(numProcs-1). For example, if numProcs is 4 and numRows is 12: Process 0 would be Master, Process 1 does rows 0-3, Process 2 does rows 4-7, Process 3 does rows 8-11,`
2) `Communication: Master divides tasks, sends exact row(s) of A to each worker, and broadcasts B. Every worker receives the broadcast of B, and where their offset in A and C is, and the data in A.`
3) `Compute: Workers compute their row(s) of C.`
4) `Barrier: Use a barrier until all workers are done.`

5) `Communication: Workers send row(s) of C and offset to master. Master receives row(s) of C and puts it into one big matrix at the correct offsets. Master can now output C.`

2) Matrix Multiplication with MapReduce: To compute matrix multiplication using MapReduce, Figure 4 shows pseudocode and how the mapping function works. Figure 5 shows pseudocode and how the reduce function works [7].

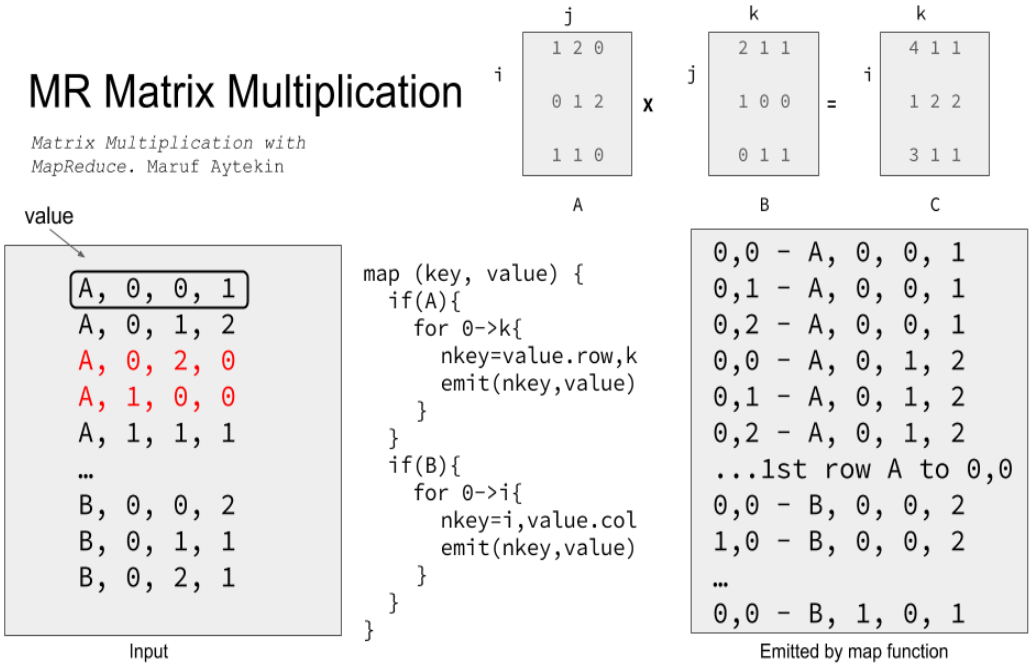


Fig. 4: Matrix multiplication map function [7].

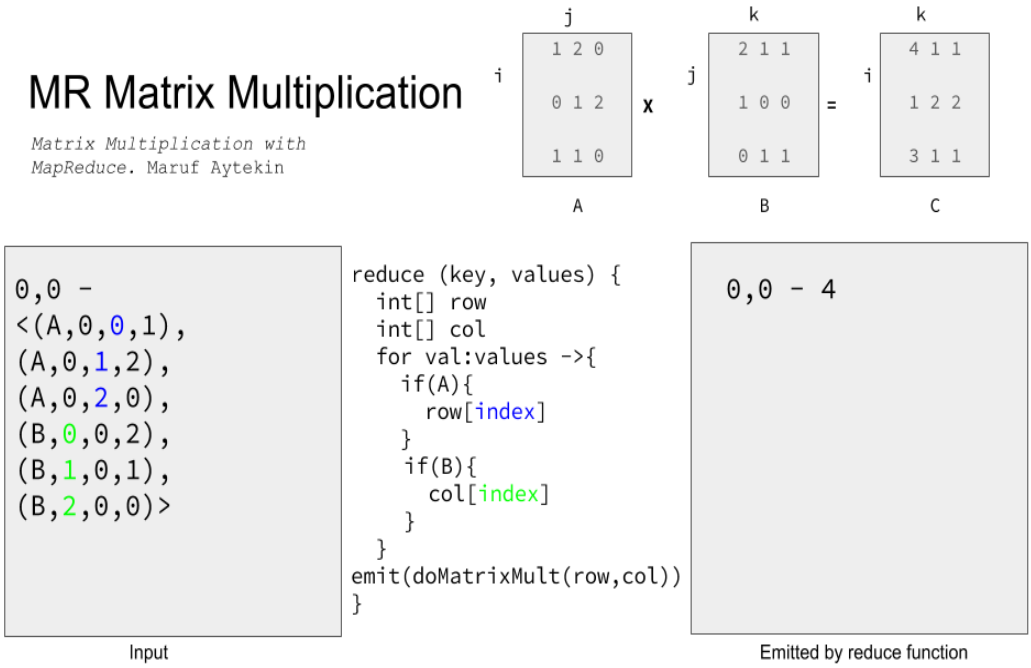


Fig. 5: Matrix multiplication reduce function [7].

For an application such as matrix multiplication, the amount of code and complexity of the code is fairly comparable between Hadoop MapReduce and MPICH2 MPI. For MPI, the user must consider how each row of the result is computed, and make sure each worker has enough information to compute that. In MapReduce, the user must consider from the perspective of the inputs, how to get to the outputs. For instance, for every cell in both input matrices, the user must know which keys in the output matrix they must map to.

The source code for matrix multiplication in MPI and MapReduce can be found at *github.com/ndhanushkodi/scientific-parallel-computing/Final*.

3) Word Count with MapReduce: To compute the number of occurences for every word in some large text file, the MapReduce pattern makes it very easy to write a program for this. Figure 6 shows how the data is moved from input to output to compute the word count.

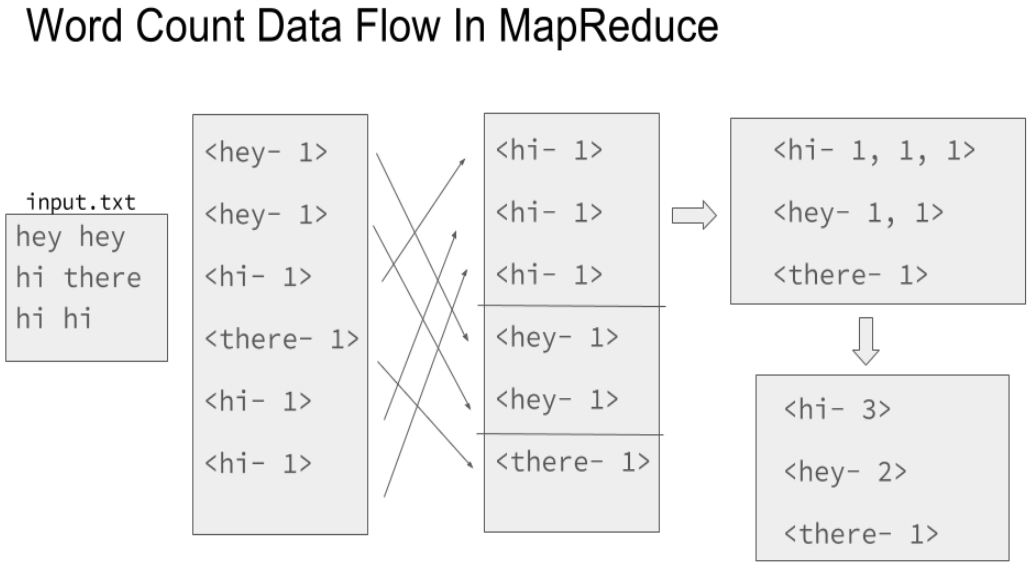


Fig. 6: Data flow for word count in MapReduce framework.

As can be seen from these examples, MapReduce works well for highly data parallel applications, where the use case has highly repetitive operations for many data across multiple machines (SPMD). For a use case such as word count, the MapReduce framework fit elegantly because of the repetitive program for lots of different data. For an example such as matrix multiplication, MapReduce and MPI are fairly comparable in terms of code complexity, although this MapReduce implementation is especially good for sparse matrices. Also, in MPI the user must decided where and how the data for an input such as a matrix should be stored. Hadoop abstracts this out with the Hadoop Distributed File System (HDFS). MPI is therefore better for smaller data, or when the user wants very different roles for different processes, as in task parallelism.

III. MapReduce Implementations

To understand the advantages and disadvantages of implementing MapReduce with MPI, first, the implementation of MapReduce with Hadoop will be discussed, then, an unoptimized implementation with MPI, and finally, a more robust implementation of the paradigm with MPI. These details will help understand the benefits and limitations of the implementations.

A. Hadoop Implementation

The book *Hadoop: A Definitive Guide*, by Tom White, describes the implementation of MapReduce in Hadoop. The MapReduce job is the work the user wants performed, and consists of the input data, MapReduce program, and configurations. Hadoop divides a job into tasks: map and reduce tasks [4].

When executing a job, there is a jobtracker node and several tasktracker nodes. The jobtracker is the node that schedules map and reduce tasks to run on tasktrackers. Tasktrackers run tasks and send updates to the jobtracker which makes sure that the tasktrackers are still running and making progress. If any tasktracker node fails, the jobtracker can schedule the task on another tasktracker [4].

The input data is stored in HDFS blocks, and blocks can reside on nodes that will be eventually running the tasks. Hadoop divides the input data into input splits, and creates one map task for each split. The map task runs the map function on each record, or value of the split. In Hadoop, the ideal split size to optimize parallelism and not create too much overhead is 64MB, which is the size of an HDFS block. Hadoop tries to optimize for data locality, which means it will try to run a map task on a node where the input split resides locally. If a split is larger than a block, it is unlikely that 2 blocks will be in the same node, so this optmization is harder. The output of the map task is written to a node's local disk, rather than HDFS. If this node does lose its data, or fails, the jobtracker will reassign the map task to another node. [4]

The reduce phase is where most of the communication between nodes happens. Each map task's results are sorted by key, and partitioned for the number of reduce tasks there are. Then the partitions are sent to the appropriate reduce task that is assigned to that key or keys. This is the communication step that the latency is much slower as implemented in Hadoop RPC versus MPI [4].

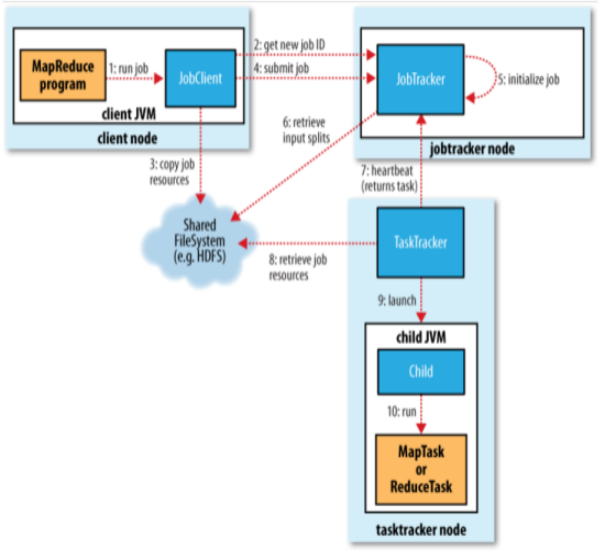


Fig. 7: This figure shows an overall picture of the roles that nodes play in a Hadoop job[4].

B. Simple MPI Implementation

To understand how MapReduce can be implemented using MPI, a simple example is considered first. As described by Hoefler et al., a simple but unoptimized implementation of MapReduce using MPI is shown[5]:

1) `Master splits up and assigns workers all the map tasks.`
2) `Once all the map tasks are finished, master splits and assigns the reduce tasks to the workers.`
3) `Each reduce task has a key and function to carry out on all values associated with that key. Each reduce task works on 1 key at a time. For each reduce task, the worker queries all other workers for the values associated with that key. These values come from the workers that carried out the map task for that key, which maps to some (key, value).`
4) `Then the worker carries out the reduce task on all of these values associated with that particular key.`

The extra functionality not already existing would be to provide a parameter called key to the reduce function. Also, the query function would have to be implemented, or each worker would have to emit to the exact process that would be reducing on that key.

Why is this unoptimized? There is a lot of communication that the master process is doing with all the workers that is not balanced with how much communication the workers do, leading to possible inefficiency. Also, when workers query keys for values from other workers, the user cannot be certain that this communication is balanced, either. Since this algorithm doesnt use the resources and communication channels in a balanced manner, more optimized MPI functions for balancing this communication should be used [5].

C. Optmized MPI Implementation

As can be seen, MapReduce implementations often use the master-worker model, and now we will see how to use collective operations to do the map and reduce tasks. The map task can use `MPI_Scatter` and reduce task can use `MPI_Reduce`[5]. Now all of the reduce tasks will be using communication via the MPI standard, which from experimental data was shown to be faster than Hadoop RPC communications [6]. Now, the reduce functions can be user defined, or the built in reduce functions can be used from the MPI library.

It is important to note there are limitations on the user-defined reduce functions and their input data [5]:

1) The `reduce` function must be associative.
2) Each process must know the number of different keys in the set K_r, or these must be communicated beforehand. The list of values v' for each key g should be able to be reduced locally.
3) If a reduce process for some key g does not have a list of values, then there must be some default, or 'identity' element supplied as the value.

Using these collective operations optimizes the implementation of MapReduce. Collective operations are optimized often even for the network hardware they are running on, or for if the number of values is small for a given key g.

IV. Results and Discussion

From the comparison section, the arguments around latency and applications will be used to identify the key differences between writing parallel programs in an implementation of MPI and writing parallel programs in an implementation of MapReduce such as Hadoop. These key differences are used to discuss the benefits and limitations of implementing MapReduce with MPI.

A. Differentiators

The key differentiators between MPI implementations and MapReduce implementations such as Hadoop include: fault tolerance, data structures storing the input data, and constraints on the reduce function if MapReduce were implemented with MPI.

B. Benefits of Implementing MapReduce with MPI

During the shuffling phase of MapReduce, when the map data sorted by key is communicated to the appropriate reduce process that would be working on that key, there is potential for a speed increase. In Figure 8, the copy phase indicates this communication over the network for the nodes performing map tasks and the nodes performing reduce tasks. As shown in the latency section in Figure 7, message latency for messages above 1MB is 100 times greater for Hadoop RPC over MPICH2. So, there is the potential for optimizing communications [6].

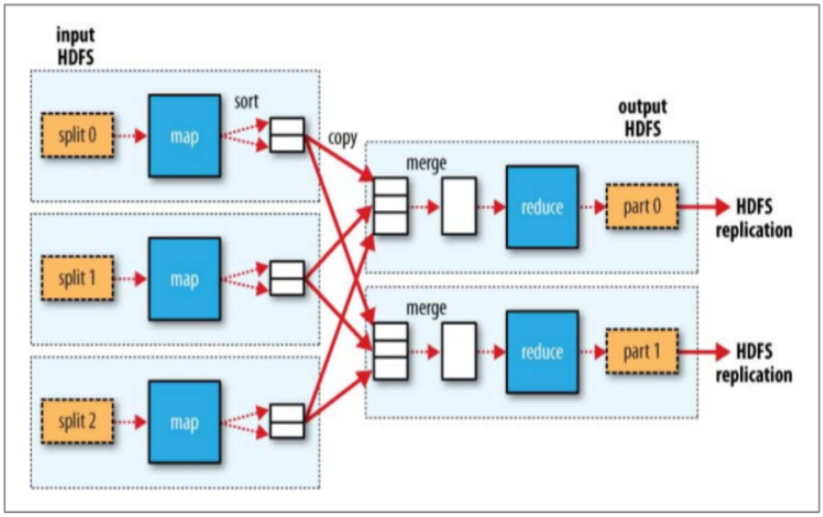


Fig. 8: This depicts the data flow in a Hadoop job [4].

C. Limitations of Implementing MapReduce with MPI

The limitations of implementing MapReduce with MPI are the following:

1) The constraints on the reduce function described in the optimized MPI implementation section.
2) Data structures are different for input data in MPI and MapReduce.
3) There is little explicit support for fault tolerance in MPI.

The data structures are different because generally, in MPI programs, the user works with fixed data structures such as matrices or arrays that have a finite length. Often these data structures are contiguously held on one node for ease of use, and this is often the master process. In MapReduce programs, the data is managed by a distributed file system and can be large and non-contiguous. The user would have to manage large and non-contiguous data when using an MPI implementation of MapReduce unless a file system was implemented with it [6].

The fault tolerance in MPI is not close to what Hadoop does for MapReduce. In MapReduce, if a task or node fails, then it is restarted. The default error handling in MPI is to abort an entire job, and if the user does handle errors, it is only handled for point to point communications, and much harder to handle in collective operations. MPI would need to internally support rebuilding communicators or restarting communications if they fail [5].

While there is potential of speedup, current implementations of MPI do not have the fault tolerance

support necessary to help implement a fully functional MapReduce, although it could be much faster. In the future, investigations of hybrid systems optimizing just small parts of MapReduce would be interesting to look at.

REFERENCES

[1] Blaise Barney. *Lawrence Livermore National Laboratory.* Message Passing Interface
https://computing.llnl.gov/tutorials/mpi/
[2] Geoffrey Fox. *MPI and Mapreduce*
http://www.netlib.org/utk/people/JackDongarra/ccgsc2010/slides/talk12-fox.pdf
[3] Sudarsun Santhiappan. *Challenges in Large Scale Machine Learning*
http://www.slideshare.net/sudarsun/challenges-in-large-scale-machine-learning
[4] Tom White. *Hadoop: The Definitive Guide*
http://ce.sysu.edu.cn/hope/UploadFiles/Education/2011/10/201110221516245419.pdf, pp28-32,167-170.
[5] Torsten Hoefler, Andrew Lumsdaine, and Jack Dongarra. *Towards Efficient MapReduce Using MPI*
http://w.unixer.de/publications/img/hoefler-map-reduce-mpi.pdf
[6] Xiaoyi Lu, Bing Wang, Li Zha, and Zhiwei Xu. *Can MPI Benefit Hadoop and MapReduce Applications?*
http://novel.ict.ac.cn/zxu/ConfPDF/Lu_ICPPw_2011.pdf
[7] *MapReduce: Matrix Multiplication*
http://hadoopgeek.com/mapreduce-matrix-multiplication/