Adaptiveness in Map-Reduce using MPI

Ahmed H.I.Lakadkutta
Mtech(CSE)
Government College of
Engineering, Amravati, India,

Pushpanjali M.Chouragade
Assistant Professor, Comp.Engg.
Government College of
Engineering, Amravati, India,

ABSTRACT

MapReduce is an emerging programming paradigm for data parallel applications proposed by Google to simplify large-scale data processing. MapReduce implementation consists of map function that processes input key/value pairs to generate intermediate key/value pairs and reduce function that merges and converts intermediate key/value pairs into final results. The reduce function can only start processing after completion of the map function. Due to dependencies between map and reduce function, if the map function is slow for any reason, this will affect the whole running time. In this technique, the message passing interface (MPI) strategies is used to implement MapReduce which reduces the runtime and optimized data exchange.MPI is used for algorithmic parallelization. MapReduce with MPI combines redistribution and reduce and moves them into the network. In this paper, new technology used as MapReduce overlapping using MPI, which is an enhancing structure of the MapReduce programming model for fast data processing. This implementation is based on running the map and the reduce functions concurrently in parallel by exchanging partial intermediate data between them in a pipeline fashion using MPI. At the same time, performing the algorithm parallelism in order to increase the performance with data parallelism of using overlapping mapreduce MPI.MPI support more efficiently all MapReduce applications.

Keywords: Hadoop, MapReduce overlapping, MPI-MapReduce, Parallel MapReduce.

I. INTRODUCTION

Today, many commercial and scientific applications require the processing of large amounts of data, and thus, demand compute resources far beyond what can be provided by a single commodity processor. To address this, the role of parallel and distributed computing becomes more important than before by offering large-scale processing capabilities. MapReduce is a programming model for data intensive applications. MapReduce was proposed by Google. It is mainly composed of two functions: a map function. MapReduce has been used to process massive amounts of data in web and a reduce function. In general, the input data is partially divided into small chunks and randomly passed to a group of map functions. The map functions process the input data and generate intermediate *(key,value)* pairs. These pairs are grouped together with respect to their *key* to produce *(key,list(values))* tuples. The tuples are then passed to a group of *reduce* functions, which do some analysis. There are many applications which can be adapted to have the same workflow including inverted indexing [1], k-means [2], sorting and PageRanking [3]. Thus, MapReduce simplifies the parallel programming process through the two *map*, and *reduce* functions. In contrast, MPI is a message passing library designed to function on parallel machines. MPI launches independent processes of an algorithm on each machine, in which the processes are connected using MPI by moving data from the address space of a certain process to another efficiently. MPI also supports collective operations, remote memory access and parallel I/O. Now, most parallel applications depend on two types of parallelization: data parallelization and algorithmic parallelization. With this two parallelism efficient communication capabilities of MPI to build a framework for fast data and algorithms processing is added using MRO-MPI model (MapReduce overlapping using MPI) as an idea to speed up the MapReduce model by avoiding its bottlenecks and using MPI functions. In the original and the current implementations of MapReduce, the reduce function has to wait for the map function to finish before it can start processing. If the map function is stuck or slow down for any reason, this will affect the whole running time as the reducers will have to wait. MRO-MPI idea is used to send partial list of intermediate data to the responsible reducer so that the reducer may start to process this partial data while the mappers continue emitting new data. Hence, the map and reduce functions works in parallel to achieve a good speedup.

In this paper MapReduce overlapping using MPI is used, because most of the current MPI-MapReduce implementations are based on the original MapReduce model.

2. ARCHITECTURE ON MAP-REDUCE

The execution of the MapReduce model is done through different stages, as follows (see also figure 1): MapReduce is a programming model as well as a framework that supports the model. The main idea of the MapReduce model is to hide details of parallel execution and allow users to focus only on data processing strategies. The MapReduce model consists of two primitive functions: Map and Reduce. The input for MapReduce is a list of (key1, value1) pairs and Map()is applied to each pair to compute intermediate key-value pairs, (key2,value2). The intermediate key-value pairs are then grouped together on the key equality basis,i.e.(key2,list(value2)). For eachkey2, Reduce()works on the list of all values, then produces zero or more aggregated results. Users can define the Map() and Reduce() functions however they want the MapReduce framework works. MapReduce utilizes the Google File System(GFS) as an underlying storage layer to read input and store output. GFS is a chunk-based distributed file system that supports fault-tolerance by data partitioning and replication. Apache Hadoop is an open-source Java implementation of MapReduce.Hadoop since Google's MapReduce code is not available to the public for its proprietary use. Other implementations (such as DISCO written in Erlang) are also available, but not as popular as Hadoop. Like MapReduce, Hadoop consists of two layers: a data storage layer called Hadoop DFS(HDFS) and a data processing layer called

Hadoop MapReduce Framework. HDFS is a block-structured file system managed by a single master node like Google's GFS. Each processing job in Hadoop is broken down to as many Map tasks as input data blocks and one or more Reduce tasks. Figure 1 illustrates an overview of the Hadoop architecture. A single MapReduce(MR) job is performed in two phases: Map and Reduce stages. The master picks idle workers and assigns each one a map or a reduce task according to the stage. Before starting the Map task, an input file is loaded on the distributed file system. At loading, the file is partitioned into multiple data blocks which have the same size, typically 64MB, and each block is triplicate to guarantee fault-tolerance. Each block is then assigned to a mapper, a worker which is assigned a map task, and the mapper applies Map() to each record in the data block. The intermediate outputs produced by the mappers are then sorted locally for grouping key-value pairs sharing the same key. After local sort, Combine() is optionally applied to perform pre-aggregation on the grouped key-value pairs so that the communication cost taken to transfer all the intermediate outputs to reducers is minimized. Then the mapped outputs are stored in local disks of the mappers, partitioned into R, where R is the number of Reduce tasks in the MR job. This partitioning is basically done by a hash function e.g., hash(key) mod R. When all Map tasks are completed, the MapReduce scheduler assigns Reduce tasks to workers. The intermediate results are shuffled and assigned to reducers via HTTPS protocol. Since all mapped outputs are already partitioned and stored in local disks, each reducer performs the shuffling by simply pulling its partition of the mapped outputs from mappers. Basically, each record of the mapped outputs is assigned to only a single reducer byone-to-one shuffling strategy. Note that this data transfer is performed by reducers' pulling intermediate results. A reducer reads the intermediate results and merges them by the intermediate keys, i.e.key2, so that all values of the same key are grouped together. This grouping is done by external merge-sort. Then each reducer applies Reduce ( ) to the intermediate values for eachkey2 it encounters. The output of reducers is stored and triplicates in HDFS. Note that the number of Map tasks does not depend on the number of nodes, but the number of input blocks. Each block is assigned to a single Map task. However, all Map tasks do not need to be executed simultaneously and neither are Reduce tasks. For example, if an input is broken down into 400 blocks and here are 40 mappers in a cluster, the number of map tasks is 400 and the map tasks are executed through 10 waves of task runs. This behavior pattern is also reported in . The MapReduce framework executes its tasks based on runtime scheduling scheme. It means that MapReduce does not build any execution plan that specifies which tasks will run on which nodes before execution. While DBMS generates a query plan tree for execution, a plan for executions in MapReduce is determined entirely at runtime. With the runtime scheduling, MapReduce achieves fault tolerance by detecting failures and reassigning tasks of failed nodes to other healthy nodes in the cluster. Nodes which have completed their tasks are assigned another input block.

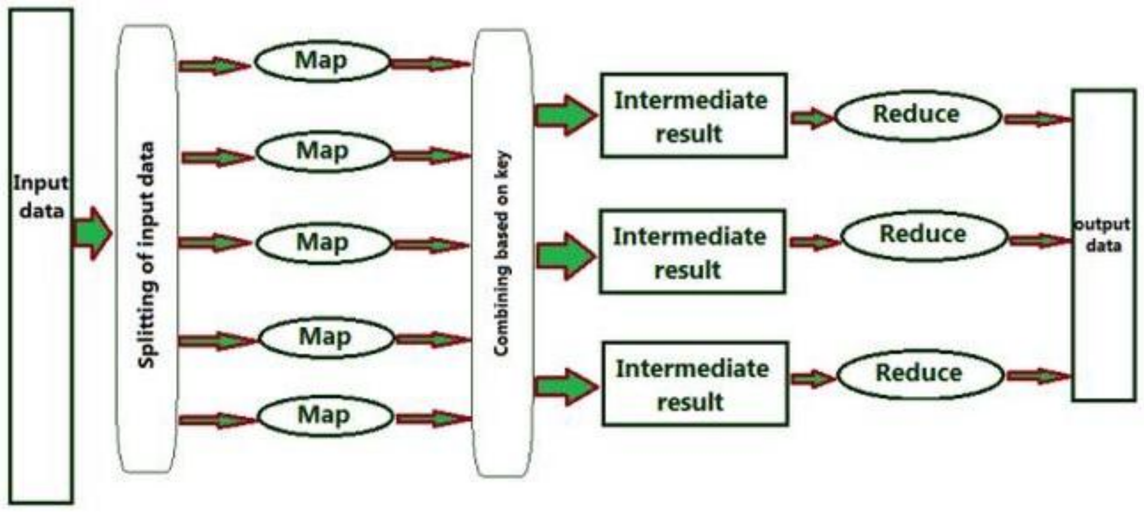


Fig 1: Map_Reduce Architecture

3. MESSAGE PASSING INTERFACE (MPI)

MPI is a library. It specifies the names, calling sequences and the results of functions to be called from C programs, subroutines to be called from Fortran programs, and the classes and methods that make up the MPI C++ library. MPI is a specification, not a articular implementation. A correct MPI program should be able to run on all MPI implementations without change. MPI is targeted towards the message passing model. Messages and buffers. Sending and receiving messages are the two fundamental operations. Messages can be typed with a tag integer. Allows message buffers to be more complex than a simple buffer and address combination by giving options to the user to create their own data types. I Separating Families of Messages. MPI programs can use the notion of contexts to separate messages in different parts of the code. Useful for writing libraries. The context are allocated at run time by the system in response to user (or library) requests.

Communicators. The notions of context and group are combined in a single object called a communicator, which becomes a argument to most point-to-point and collective operations. Thus the destination or source specified in send or receive operation always refers to the rank of the process in the group identified by a communicator. For example, consider the following blocking send and blocking receive operations in MPI. MPI_Send(address, count, data type, destination, tag, comm) MPI_Recv(address, maxcount, data type, source, tag, comm, status) The status object in the receive holds information about the actual message size, source and tag.

Collective Communications. MPI provides two types of collective operations, performed by all the processes in the computation.

Data movement: Broadcast, scattering, gathering and others.

Collective computation: Reduction operations like minimum, maximum, sum, logical OR etc as well as user-defined operations.

Groups: Operations of creating and managing groups in a scalable manner. These can be used to control the scope of the above collective operations.

Debugging and profiling. MPI requires the availability of "hooks" that allow users to intercept MPI calls and thus define their own debugging and profiling mechanisms.

Support for libraries. Explicit support for writing libraries that are independent of user-code and inter-operable with other libraries. Libraries can maintain arbitrary data, called attributes, associated with the communicators they allocate and can specify their own error handlers.

Support for heterogeneous networks. MPI programs can run on a heterogeneous network without the user having to worry about data type conversions.

Processes and processors. The MPI specification uses processes only. Thus the mapping of processes to processors is up to the implementation.

Table I Example of Most Common MPI Functions

MPI Method	Description
MPI Send()	Send data directly to a certain process
MPI Recv()	Receive data from a certain process
MPI Gather()	Gather data from different processes
MPI Scatter()	Scatter data on the processes
MPI Bcast()	Broadcast data on all processes

4. MRO-MPI Model

In MPI, the sender has to define the rank of the process that receives the data and the type and the size of the sent data. At the same time, the receiver has to be ready and informed about the received data. Thus, this decreases the usability of MapReduce using MPI. Figure 3 shows the architecture of discussed prototype. The Map and Reduce sides are divided into two parts; user and system. The user side is the part where the programmer writes the mapping function. The system side is the part which is responsible to handle the communication.

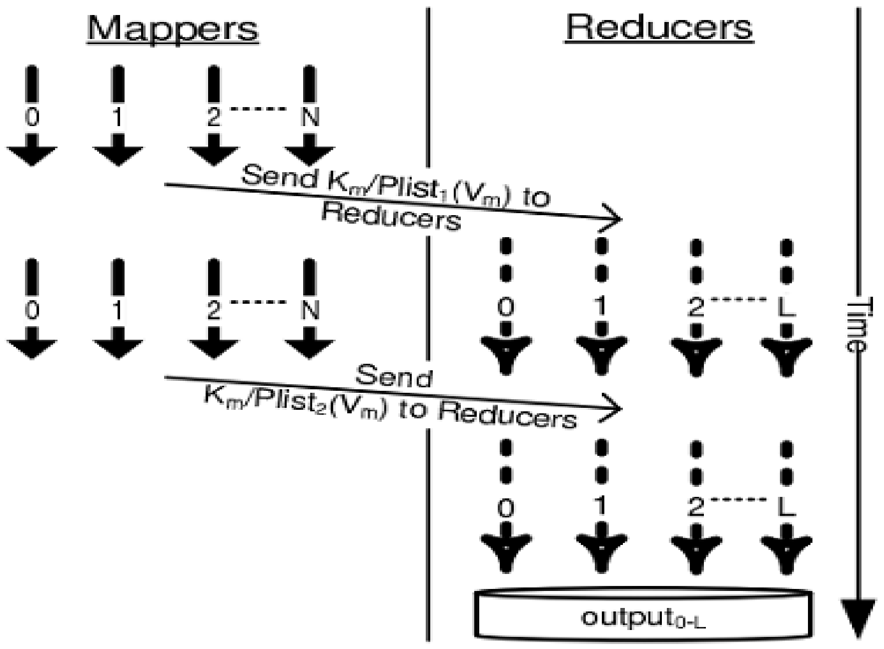

Mappers
Reducers
0
1
2
N
Send $K_m/Plist_1(V_m)$ to Reducers
0
1
2
N
0
1
2
L
Time
Send $K_m/Plist_2(V_m)$ to Reducers
0
1
2
L
$output_{0-L}$

Fig 2. MRO-MPI: The mappers and reducers work in parallel and partial data is sent in a pipeline fashion.

Advanced model is based on three steps as follows:

Mapping and Shuffling :In this model, the mapping and the shuffling are merged in one phase. The mapping is exactly like the original one. The map functions emit (Km,Vm) pairs. The Km of each pair is passed to a partitioning function.

Partitioning:(Km) → (hkl), the output range of the function is based on the number of the reducers; for L reducers the range is : 0→L. The default function is a”Hash”function, which is similar to hash Partitioner in Hadoop, but also it can be replaced by user-defined hash function.

Map step: Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node orchestrates that for redundant copies of input data, only one is processed.

Shuffle step: Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.

Logical View: The Map and Reduce functions of MapReduce are both defined with respect to data structured in (key, value) pairs. Map takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

$$\text{Map(k1,v1)} \rightarrow \text{list(k2,v2)}$$

The Map function is applied in parallel to every pair in the input dataset. This produces a list of pairs for each call. After that, the MapReduce framework collects all pairs with the same key from all lists and groups them together, creating one group for each key.

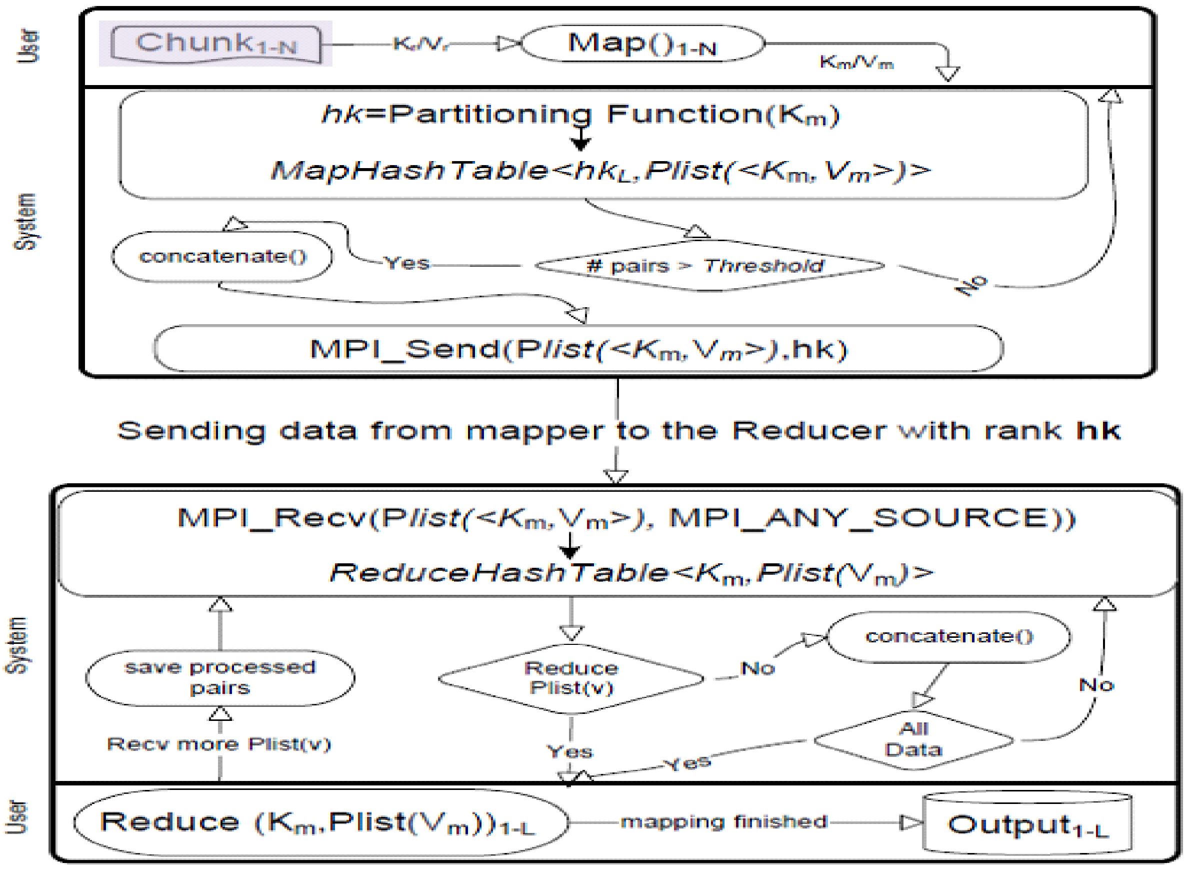


Figure 3. MRO-MPI: Technical details and data merging

For the default hash function, normally hash collisions occurs as the number of keys is more than the number of the reducers, but it happens with equal distribution to roughly make equal load balancing of the keys on the reducers. The (hkl,(Km,Vm)) pairs are saved in a local hash table for each process. Eachhkl is associated with a list of various (Km,Vm) pairs:

MapHashtable :
(hkl)→((Ki,V1),(Ki,V2),(Ki+1,V1),...(Km,Vm))

There are l counters C0−l, each counter is assigned to a hash key hkl. They are used to count the size of pairs associated to eachhkl. Every time a new pair is emitted, counter Chkl is checked if it is greater than a threshold value T, which is user-defined. If so, the partial data is concatenated as one chunk and sent directly to the responsible reducer which has a rank hkl. Sending Data in MPI is based on the MPI Data types like MPIINT, MPI CHAR,... etc. Additionally, MPI gives the user the ability to construct his own data types based on the original ones, which is called “derived data types”.

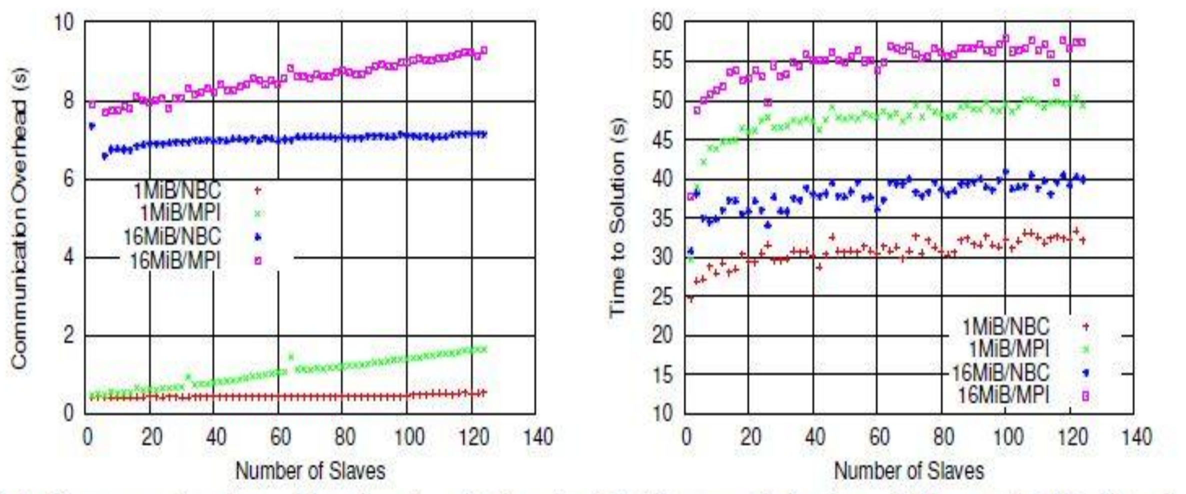
Receiving and Merging: All Reducers are actors, which means that they are ready to receive data from any mapper. The received data contains multiple intermediate (Km,Vm) pairs. A hash table is responsible for organizing this data. The key of this hash table is the intermediate key and the value is a list of intermediate values received in this partial list and related to this key:

Reduce Hash Table: Km→P list(Vm).

Reducing: After grouping the data, the system checks if the user asked for reducing partial data or not, as in some applications, the reduce function needs the full (Km, list(Vm))(e.g. max or average functions). In this case, the reduce function receives the data and merges it with the already saved data without any processing until the complete mapping is done. If partial reduce can be processed (e.g. sum functions), all the keys within its partial list are passed to the reduce function one by one. After reducing, the partial data is saved in local memory. For the two scenarios, after saving the data, the system calls the receiving function again and this process continues until all the data is received from the mappers. When mapping is done and last partial data are reduced, output data are saved on the local hard disk of each reduce process. Hence in this model,four phases are truncated into three phases. The three phases run in parallel on different machines and continue until the mapping is done. The user has to define the number of mappers, reducers, and the threshold valueT. The ratio between the mappers and the reducers affects the performance of the model. A good ratio between the mappers and reducers with analysis and the effect of changing the T value are given in the next section.

5. PERFORMANCE RESULTS

MapReduce applications typically process large amounts of data that have to be read from either the network or local disks. Thus, we assume that the I/O bandwidth is not sufficient to keep multiple processing elements busy. However, most of today's systems are multi-core or SMP systems such that there are idle cores available to offload the communication. We use the threaded InfiniBandoptimized version of LibNBC [17,18] for all benchmarks. This efficiently results in offloading the reduce task to another core (the reduce operation is a part of the NBC Reduce communication) and thus utilizes another level of functional parallelism transparently to the application developer. Benchmarks of the simple string-search example were also covered by the more extensive simulator and delivered exactly the same results. Thus, we only present benchmark results for the different configurations of the simulator. We benchmarked two different workload-scenarios with 1 to 126 worker nodes with 10 tasks per process. We compared the threaded version of LibNBC with a maximum of 5 outstanding collective operations with Open MPI 1.2.6. We also varied the data-size of the reduction operation (in our example, we used MPI SUM as the reduction operation). Figure 4(a) shows the communication and synchronization overhead for a static workload of 1 second per packet. Using nonblocking collective results in a significant performance increase because nearly all communication can be overlapped. The remaining communication overhead is due to InfiniBand's memory registration which is done on the host CPU. The graphs show a reduction of communication and synchronization overhead of up to 27%. Figure 4(b) shows. the influence of nonblocking collectives to dynamic workloads varying between 1ms and 10 s. The significant performance increase is due to avoidance of synchronization and the use of communication/computation overlap. This clearly shows that our technique can be used to benefit MapReduce-like applications significantly. The dynamic example shows improvements in time to solution of up to 25% over the unoptimized implementation.



(a) Communication Overhead of Static Workload (b) Time to Solution of Dynamic Workload

Fig. 4. Overhead and Time to Solution for Static and Dynamic Workloads for different Number of Workers

6. CONCLUSIONS

MapReduce and MPI were developed in two different communities that have traditionally been somewhat disjoint. However, as the needs and capabilities of these two communities continue to converge, it will be to the benefit of both to leverage their respective technologies. In the case of MapReduce and MPI, it is possible to efficiently implement MapReduce using MPI – with some limitations. For example, HPC-centric optimizations can be applied if the reduce function fulfills certain criteria. Additional performance gains are possible through upcoming MPI features. Using nonblocking collective operations, for example, provided a speedup of up to 25% over the blocking implementation. Fully supporting MapReduce will require several additional features and capabilities from MPI. However, many of these features are generally recognized as being important, particularly as MPI evolves to support other modern programming and parallelization paradigms

7. REFERENCES

[1] T. White, Hadoop: The Definitive Guide, first edition ed. O'Reilly, june 2009.

[2] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "Mapreduce with communication overlap," in Technical 2007.

[3] M. Elteir, H. Lin, and W. chun Feng, "Enhancing mapreduce via asynchronous data processing," in Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on, dec. 2010, pp. 397 –405.

[4] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards efficient mapreduce using mpi." In PVM/MPI, ser. Lecture Notes in Computer Science, M. Ropo, J. Westerholm, and J. Dongarra,Eds., vol. 5759. Springer, 2009, pp.240–249.

[5] M. Elteir, H. Lin, and W. chun Feng, "Enhancing mapreduce via asynchronous data processing," in *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, dec. 2010, pp. 397 – 405

[6] Hishan Mohamed, Stephane´ Marchand-Maillet, "Enhancing MapReduce using MPI and an optimized data exchange policy" in 2012 41st International Conference on Parallel Processing Workshops

[7] Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters.Commun. ACM 51(1).

[8] L¨ammel, R.: Google's MapReduce programming model — Revisited. Sci. Comput.rogram. 68(3) (2007) .

[9] de Kruijf, M., Sankaralingam, K.: MapReduce for the CELL B.E. Architecture.IBM Journal of Research and Development 52(4) (2007)

[10] He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a MapReduce framework on graphics processors. In: PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, New York, NY, USA, ACM (2008) 260–269

[11] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for Multi- core andMultiprocessor Systems. In: HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, Washington, DC, USA, IEEE Computer Society (2007) 13–24

[12] Langville, A.N., Meyer, C.D.: Google's PageRank and Beyond: The Science of Search Engine Rankings. Princeton University Press (July 2006)

[13] Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G.R., Ng, A.Y., Olukotun, K.:Map-Reduce for Machine Learning on Multicore. In Sch¨olkopf, B., Platt, J.C., Hoffman, T., eds.: NIPS, MIT Press (2006) 281–288

[14] Kimball, A., Michels-Slettvet, S., Bisciglia, C.: Cluster computing for web-scale data processing. SIGCSE Bull. 40(1) (2008) 116–120

[15] Hadoop: http://hadoop.apache.org (2009)

[16] Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: Parallel analysis with Sawzall. Scientific Programming 13(4) (2005) 277–298

[17] Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. SIGOPS Oper.Syst. Rev. 37(5) (2003) 29–43