Research Article

Vignesh Srinivasakumar, Muthumanikandan Vanamoorthy*, Siddarth Sairaj, and Sainath Ganesh

An alternative C++-based HPC system for Hadoop MapReduce

https://doi.org/10.1515/comp-2022-0246
received June 26, 2020; accepted June 17, 2022

Abstract: MapReduce (MR) is a technique used to improve distributed data processing vastly and can massively speed up computation. Hadoop and MR rely on memory-intensive JVM and Java. A MR framework based on High-Performance Computing (HPC) could be used, which is both memory-efficient and faster than standard MR. This article explores a C++-based approach to MR and its feasibility on multiple factors like developer friendliness, deployment interface, efficiency, and scalability. This article also introduces Eager Reduction and Delayed Reduction techniques to speed up MR.

Keywords: MapReduce, machine learning, VM cluster, OpenMP, OpenMPI, containers, JVM, HPC

1 Introduction

MapReduce (MR) is a programming paradigm for large scale data processing. Logically, each MR operation consists of two phases: a map phase where each input data is mapped to a set of intermediate key/value pairs, and a reduce phase where the pairs with the same key are put together and reduced to a single key/value pair according to a user specified reduce function. This is an efficient way of parallel computing across distributed computers for big data tasks. The caveat is that the entire ecosystem revolves around JVM which was once a viable solution as it could run on any hardware and did not run into memory and garbage allocation problems.

The original focus of Hadoop was data storage and processing, which means jobs are submitted and processed in batches which were very slow due to its dependence on disk. Over the last few years, the scenario has changed drastically as Big Data analytics has taken common ground where MR jobs have become more real time since the need for faster outputs has become quintessential.

There are Pros and Cons of Java in using the Hadoop Ecosystem [1]. Java was originally adopted for Hadoop and sequentially MR to cooperate with the original Nutch framework for search engines. This method poses the following advantages:

- Detailed debugging experience.
- Mature ecosystem of developers and tools.
- Type safety and garbage collection could prevent memory leaks.
- Java is compiled to byte code for the JVM, which works on any system. Hence portable.

Java comes along with its own fair share of disadvantages like [2]:

- Memory overhead is a real problem when a JVM uses large amounts of memory just to persist, which could have been used for a computational task.
- Data Skew in Hadoop's MapReduce is a real problem when some parts of the workload are completed before others [3].
- Java's implementation of data flows, i.e., de-serialization and uncompressing of records from storage is very slow due to the creation and deletion of too many objects.
- Bindings is not generally possible to interface directly with Java from another language, unless that language is also built on top of the JVM. There are many problems in Hadoop that would be better solved by non-JVM languages.
- The decreasing popularity of Java is due to its high verbosity, which prevents user adoption and rapid prototyping.

* **Corresponding author: Muthumanikandan Vanamoorthy,** School of Computer Science and Engineering (SCOPE), Vellore Institute of Technology, Chennai, Tamil Nadu, India,
e-mail: muthumanikandan.v@vit.ac.in
Vignesh Srinivasakumar: School of Computer Science and Engineering (SCOPE), Vellore Institute of Technology, Chennai, Tamil Nadu, India, e-mail: s.vignesh2017a@vitstudent.ac.in
Siddarth Sairaj: School of Computer Science and Engineering (SCOPE), Vellore Institute of Technology, Chennai, Tamil Nadu, India, e-mail: siddarthsairaj2017@vitstudent.ac.in
Sainath Ganesh: School of Computer Science and Engineering (SCOPE), Vellore Institute of Technology, Chennai, Tamil Nadu, India, e-mail: Sainath.g2017@vitstudent.ac.in

- Iterative and incremental processing is much more difficult.
- For real time results, most modern big data computations rely on memory and faster computation speeds, making C++ a better alternative to Java.

Hence, it is clear that we need to explore other options that promote faster processing.

The rest of this article is organized as follows. Section 2 mainly reviews some existing techniques for implementing a C++-based framework in the Hadoop environment. Section 3 presents formal definitions and pseudocode for the model proposed. Section 4 highlights the details of possible implementations through a custom device devised for experimenting with the model. Section 5 describes the implementation and results obtained. Finally, we conclude this article in Section 6.

2 Related works

There are few other works that aim to perform the same essential task, each with their own merits and demerits. The key contributions of refs [4–6] are a simple and powerful interface for seamless parallelization and distribution of large-scale calculations, as well as MR implementation of this interface that delivers excellent performance on large clusters of commodity PCs. Ref. [4] presented a thorough explanation of a MR interface implementation suitable for cluster-based computing environments [5] The general architecture for the use case scenario on business process management, based on the popular MR platform Hadoop, was presented, with an emphasis on the procedures for monitoring the MR apps and conducting cluster recovery operations. Ref. [6] introduces the MR programming model and the standard configuration for running Hadoop on the public cloud, as well as an examination of the MR program execution. In its work, it also provides the cost model as well as choice issues on resource provisioning, as well as various optimization problems based on the cost model. The usage of SDN in handling link failures is discussed which also has the possibility to integrate MR framework [7].

The MR-MPI [8] implementation is an open-source C++ framework which provides a C wrapper for Python. The framework works in three interface steps: Map (), collate (), and reduce () but the actual process is more like: map → aggregate → convert and then finally reduce. User provides callback functions to implement the map and reduce phase while MR-MPI contains functions to perform aggregate and convert. The map () is a user-defined function which generates a page-memory worth of key-value (KV) pairs. An MPI_Alltoall () method is used to communicate the pairs to other processors. The collate () converts the KV pairs into a list of values per key. It can also operate out-of-core, which means that it can fit data out of memory by writing temporarily to disk, which does not exceed seven files. It also sorts it based on keys in O ($N\log N$) time using Merge-Sort. The reduce () converts the KV lists into reduced data. The framework is especially useful for graph-based algorithms. It proposes an enhanced algorithm that works well with the given framework. The graph algorithms like R-MAT, single source shortest path, maximally independent sets, triangle finding, connected components, and PageRank were tested on medium-sized Linux clusters and compared with MR implementations. It has other useful features, like in-core processing for data that fits in memory and out-of-core processing for larger datasets. The biggest disadvantage is that it is not fault tolerant, which is caused by MPI. Current MPICH has improved the state of fault tolerance in MPI using Hydra for error reporting. The other conclusions that were drawn are that randomization of data across processors eliminates data locality but is efficient for load-balancing on even irregular data, this framework can be exploited to maintain processor specific "state," which would not have been possible in cloud-based Hadoop, and that MR based on MPI is easier to code and scale.

Mimir [9] is another optimization and execution framework for better MR-MPI. Mimir takes the MR-MPI implementation as a base but significantly improves performance and reduces memory load using the proposed optimization techniques to reduce memory. The article claims that MR-MPI uses global barriers to synchronize data at the end of each phase, which demands that all the data from the current phase be retained in memory or on the input/output (I/O) subsystem until the next phase starts. This intermediate data can be very large for iterative MR tasks. One of the problems of IBM-like supercomputers is that they use a lightweight kernel that does not handle memory fragmentation due to frequent allocation and deallocation of memory buffers of different sizes. MR-MPI uses "pages" to store intermediate data. When "page" memory is full, it spills data via I/O to disc in an out-of-core fashion. This poses as a potential bottleneck as supercomputers do not generally have local discs and when I/O wants to write, it writes on a global parallel filesystem and this makes disk-spilling expensive. MR-MPI also suffers from redundant memory buffers and unnecessary memory copies. As a result, the article implements a system for significantly more memory-efficient

MR-MPI implementation. It does so by introducing two more objects, called KV containers and KMV containers, to help manage MR-MPI's KVs and KMVs (key-value merged lists).

Blaze [10] is a similar standard for an MP/MPI based MR. It is a modern C++ 17-based framework that was introduced by students at Cornell University. It brings features like Eager reduction, Thread local cache, and Fast-serialization. Other MPI implementations use ProtoBuf by Google to serialize and deserialize data before transmitting across processors, but it performs much better than the other libraries but has its own shortcomings. Eager reduction speeds up execution, but there is no alternative for classic MR-based algorithms. Eager reduction is a feature of Blaze [10] that speeds up the reduction process by reducing as soon as the map process is done while shuffling.

Figure 1 explains how a classic MR job is processed. First is the mapping phase, where the input is passed through mappers on the cluster nodes in parallel. Then, comes the shuffle phase, where the outputs of the map phase are transmitted across the network to the assigned reducer. Then, occurs the reduce phase, which performs the reduce operation on the accumulated results of all the mappers from all the cluster nodes.

Figure 2 explains the working of Blaze's MR where the shuffle and reduce occur simultaneously. Reduce is applied to the output of the mapper locally at the MPI slave level and then simultaneously shuffled across the network for the final shuffle phase. There is a Thread local cache that reduces the movement of data across processors and increases execution speed by caching.

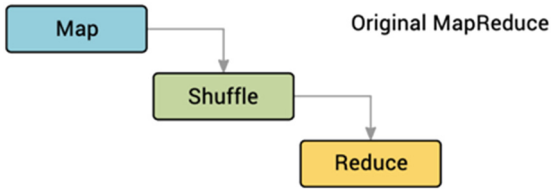

Figure 1: Classic MapReduce pattern.

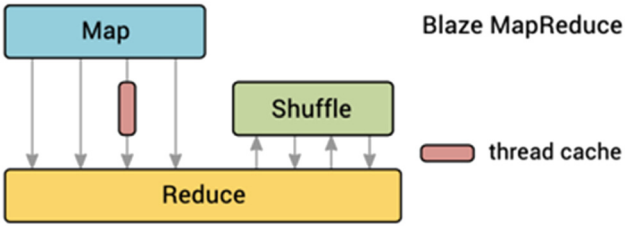

Figure 2: Blaze MapReduce pattern with Eager reduction.

Mariane [11] is another High-Performance Computing (HPC)-based MR implementation. It is an MR implementation adapted and designed to work with existing and popular distributed file systems. It can handle various clusters, shared-disks, and POSIX parallel filesystems. It eliminates the need for a HDFS or other dedicated filesystem that needs to be maintained for a MR system's purpose. It is said to work with NERSC, the Open Science Grid, NY State Grid, TeraGrid, and other HPC grids based on MPI. In its implementation, it is prescribed to use I/O management and distribution rests within the Splitter. Concurrency management is handled by the Task Controller, while fault tolerance is handled by the Fault Tracker. In terms of input management, Mariane requires the underlying filesystem to take care of input file splitting and input distribution by utilizing the "elasticity" of cloud and other distributed storage platforms. For task management, Mariane implements a Task Tracker maintained by a master node, which monitors subtasks using a task completion table. If a task fails, the Fault Tracker reassigns the job based on file markers, unlike Hadoop, which is based on input splits. Hence, achieving higher performance and speeds without compromising fault tolerance is crucial.

The paper [12] is a benchmark to test the performance of OpenMP, MPI, and Hadoop for MR tasks. It misses out on one key aspect: that OpenMP can be run on nodes individually on an MPI cluster, which means that on each MPI node, some of the tasks can be executed in parallel along with being distributed. It states that OpenMP is easier as programmers do not have to consider workload partitioning and synchronization. It also states that "MPI allows more flexible control structures than MR"; hence, it states MPI as a good choice when a program needs to be executed in a parallel and distributed manner with complicated coordination among processes.

Microsoft Azure is another cloud-based platform that provides computing resources and has its own proprietary MR implementation too [13].

Amazon Web Services provide an Elastic Cloud Platform that implements its own Amazon Elastic File System and has its own proprietary MR [14,15].

Smart [16] is a MR-like system for *in situ* data processing on supercomputing systems. Smart deviates from the traditional MR to be much more adept at meeting the needs of *in situ* data processing and not provide all of the MR semantics and interfaces.

Mimir+ [17] is a further optimized version of Mimir for GPU-accelerated heterogeneous systems. It was tested on the Tianhe-2 supercomputer and it outperformed Mimir on data-intensive tasks. They proposed a pre-acceleration

system that works before the GPU is used and performs operations like data partitioning, data communication, and data transmission.

Ref. [18] provides an interface for using MPI on Docker and using a separate registry for the containers to access files using MPI on a Docker swarm.

The necessary steps needed to set up a Raspberry Pi Beowulf cluster that would be needed to build and test the libraries discussed in ref. [19] for small-scale hardware tests.

The paper [20] provides the algorithm to implement the parallel *K*-means algorithm using iterative MR on a distributed cluster of machines. This is later implemented in Blaze.

DELMA [21] is a proposed framework that sheds perspective on a dynamic set of nodes that can be scaled up and down without interrupting the current executing jobs. The paper provides compelling reasons for a MR framework.

3 Proposed system

To use a HPC system as a MR in production or on massively parallel systems, one of the three approaches can be adopted.

3.1 Bare metal hardware

The most common configuration for most Distributed Computing Clusters is commodity hardware. Most DFS clusters run on commodity hardware. The following are installed: MPICH, OpenMP, and SSH.

Figure 3 describes a common setup for HPC applications where Master and Slave nodes communicate via MPI using OpenSSH. Each node also processes in parallel using OpenMP individually. SSH key exchange and password-less SSH are required for the communication to take place.

3.2 VM cluster-based MR

Figure 4 explains a better and more efficient setup for clusters by using VMs. This is especially advantageous as it reduces the cost of setting up and reduces hassle. It offers other advantages such as environment isolation, easier recovery, faster updates to environment, and easier maintenance. The only disadvantage is the overhead imposed by hypervisor which causes increased boot up times and slower performance on some instructions.

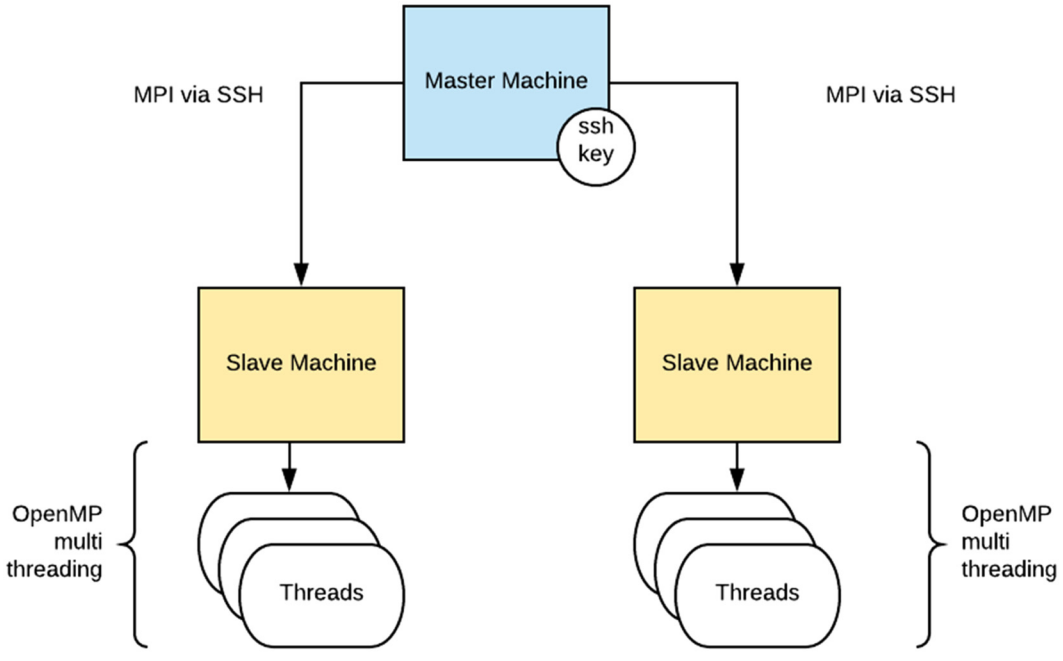

Figure 3: Architecture for Bare metal MPI cluster with OpenMP.

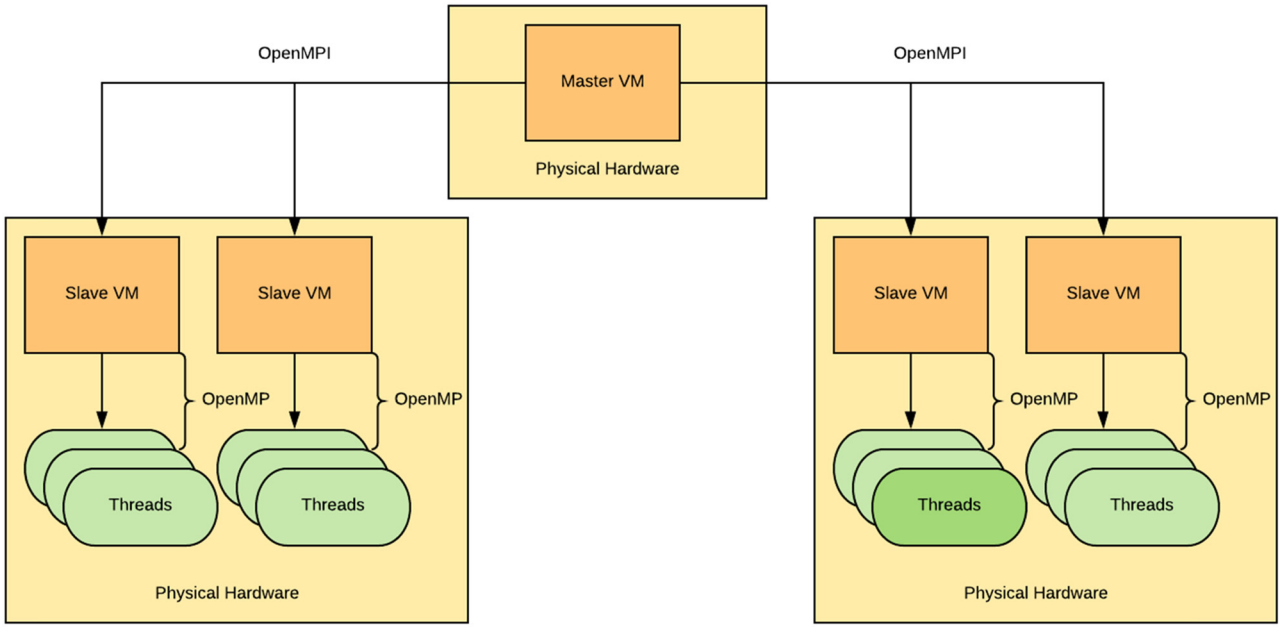

Figure 4: Architecture for VM-based MPI cluster with OpenMP.

3.3 Containerization-based MR

The paper proposes a Docker or Singularity based containerized application. The advantages of containerizing are many. The first and foremost important aspect is portability. Apart from portability, they provide efficient utilization of hardware and with orchestration methods such as Kubernetes, it becomes easier to recover from failure and maintain ready resources and application states.

Figure 5 explains a container-based MPI clusters where a Docker service for SSH has been separately deployed and used for communication services. The MPI slaves are containers created from MPI slave images while the master is

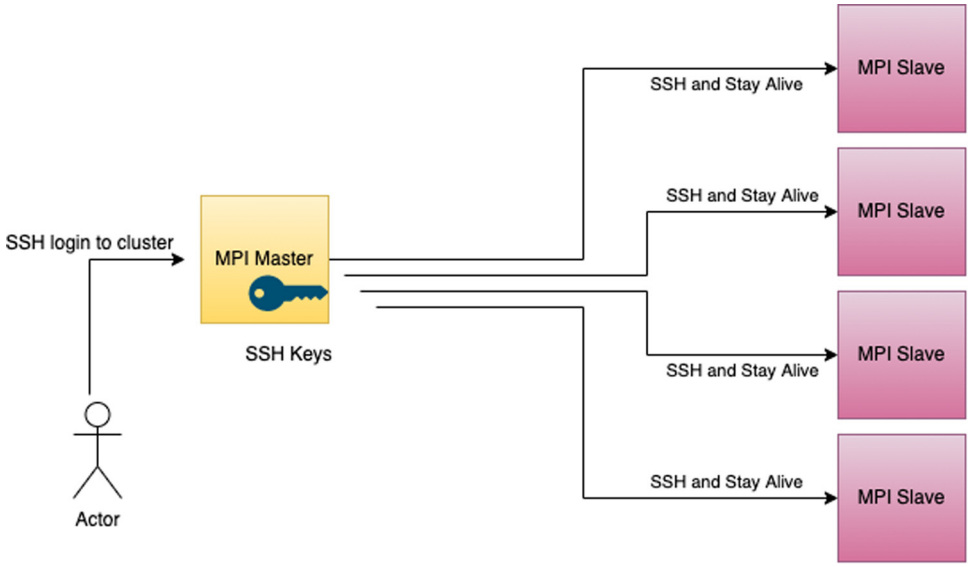

Figure 5: Architecture for container-based MPI cluster with OpenMP.

from the master image. During the automated setup using docker-swarm utility, the SSH keys are copied from master image to slave image and an endpoint is provided for the user to communicate to the master container. This type of MPI cluster setup is massively advantageous as it provides fault tolerance, faster setup, better utilization of resources, portability, and efficient updates to software. In contrast to the VMs, containerized approach has negligible overhead.

3.4 Delayed reduction

When the framework was used to develop other algorithms like matrix multiplication and linear regression, it felt rigidity due to the eager reduction and it was almost impossible to implement the algorithms.

This becomes an issue when reduction has to be done over the iterable list of HashMap which currently is not possible in Blaze framework. Hence, to solve this issue, Delayed reduction has been added to framework as shown in Figure 6.

To implement Delayed reduction, we need to understand two APIs provided by Blaze: DistVector and DistHashmap.

- A DistHashmap works by sharding, balancing, and distributing a standard C++-based HashMap over an MPI cluster using collective communication.
- The DistVector is based on DistHashMap but with serial keys that converts a C++ standard vector into a DistHashMap and then shards, balances, and distributes across the MPI cluster.

In eager reduction, the reduction occurs as soon as part of mapper process completes between two values with the same key. As an alternative to this, Delayed Reduction works by creating a temporary DistVector and emitting it as the output of a mapper which contains all the locally reduced values. This DistVector is reduced immediately into another DistVector after sorting using Merge Sort and then shuffled across the network. The final reducer does the actual reduction job but now it operates on an Iterable of Values instead of a single value, as would have been possible with Eager reduction.

The Figure 7 can be more formally represented by the following pseudocode.

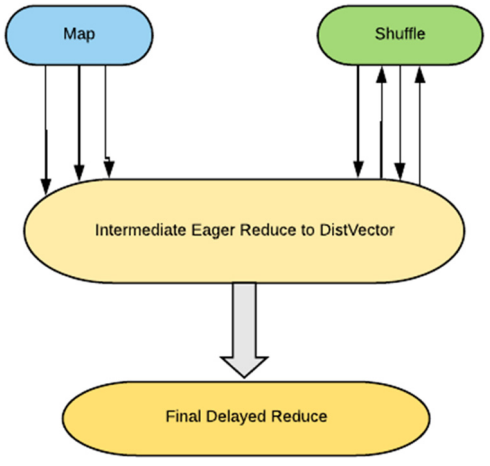

Figure 6: Improved blaze MR pattern with Delayed reduction.

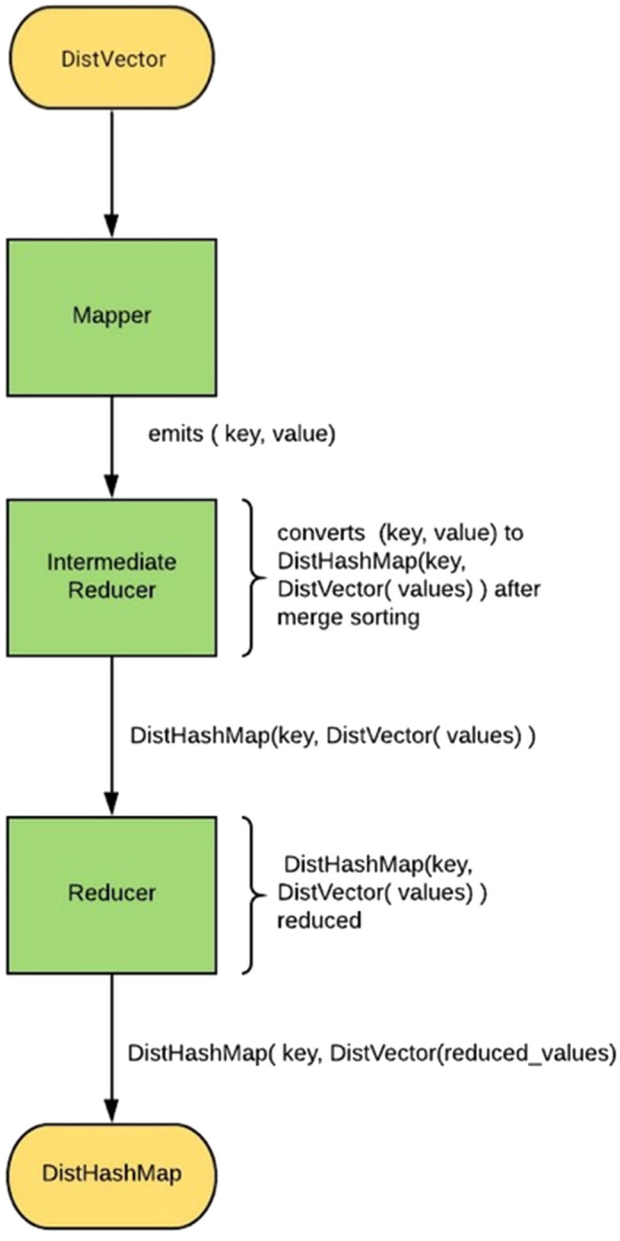

Figure 7: Algorithm for delayed reduction.

3.4.1 Pseudocode

1) A DistVector or DistHashMap or a C++ STL vector contains the source for MR.
2) Mapper can be any function that emits a KV pair and acts on the Source.
3) Intermediate reducer combines the keys into a DistVector.
4) MR is called on the source DistVector to convert it into a Key and Iterable <Value>. This DistVector is distributed across the cluster in-memory.
5) The final Reducer works on an Iterable of Values now. This can be called immediately or later. Laziness of Reduction is displayed.
6) The final DistHashMap is used to hold final Reduced HashMap in a distributed manner.

4 Specific details about potential implementations

There are three ways an HPC system-based MR framework can be used in increasing scale:

- Virtual machines that are configured to run on hardware.
- Directly on commodity grade hardware.
- Containerized images using Docker or Singularity.

The steps to setup and configure a working cluster on the three different hardware is as follows.

4.1 Raspberry pi MPI cluster

The experimental setup that was used to simulate commodity hardware was Raspberry pi 3B + with 1GB LPDDR2 SDRAM, Gigabit Ethernet, and MPICH2. Steps to setup an MPI cluster on an array of Raspberry Pi are as follows:

Install Raspbian OS and install physical ethernet network. Dedicate one RPI as master and the rest as slaves. Install MPICH2 on all devices. Enable password-less SSH from master to all the slaves. Create Hostfile with all the IP addresses of the slaves. Mpirun the code with the library in path, along with hostfile each time.

4.2 VM MPI cluster

The Ubuntu 18.04 VMs with configurations of 4 GB RAM, 10 GB SSD, VirtualBox images on Ubuntu 18.04, and OpenMPI 2.1.1 are used. Steps to setup an MPI cluster on an array of VirtualBox VMs:

Create an Ubuntu 18.04 VM and allocate resources and assign a static IP on a Bridge Network. Install OpenMPI and accompanying libraries on the created VM. This VM will be the master node. Make clones of the created VM for slave nodes. Update to unique IPs of the slave VMs. Enable password-less SSH from master to all the slaves. Create Hostfile with all the IP addresses of the slaves. Mpirun along with hostfile each time.

4.3 Containerized application using docker

The steps to set up containerized is as follows using the docker image from ref. [22]:

Install Docker daemon on host using *$ sudo apt-get install docker*. Pull nlknguyen/alpine-mpich from Docker hub using the command *$ docker pull nlknguyen/alpine-mpich*. Use this command to run an MPI image for development, $ *docker run –allow-run-as-root –rm -it -v '$(pwd):/project' nlknguyen/alpine-mpich*. Use the existing Docker-swarm configuration to create an MPI cluster. *$./cluster/cluster.sh up size = 2*. Use this command to execute any MPI run command. *$./cluster/cluster.sh exec <command>*.

5 Implementation and results

The library of choice to establish a complete MR system was Blaze [23], a modern alternative to Google MR-MPI with features like eager reduction, thread local cache, and fast serialization which has the potential to boost performance of MR algorithms. The problem caused by the lack of fault tolerability still persists due to the MPI implementation.

A custom test was devised to measure aspects on real hardware for testing performances in smaller key ranges and datasets for performance, scalability, and adaptability for different algorithms.

For larger datasets, it is compared against the Spark implementation of the algorithms using MLlib library that implements machine learning algorithms.

5.1 *K*-Means clustering using MR

K-means clustering is an unsupervised machine learning algorithm for classification. This can be implemented in MR using the algorithm described in ref. [20].

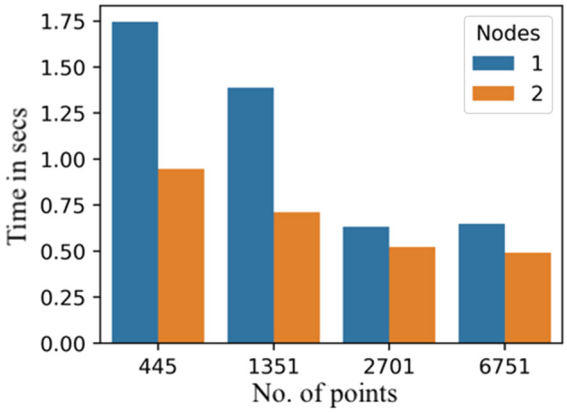

Figure 8: *K*-means clustering on Blaze framework.

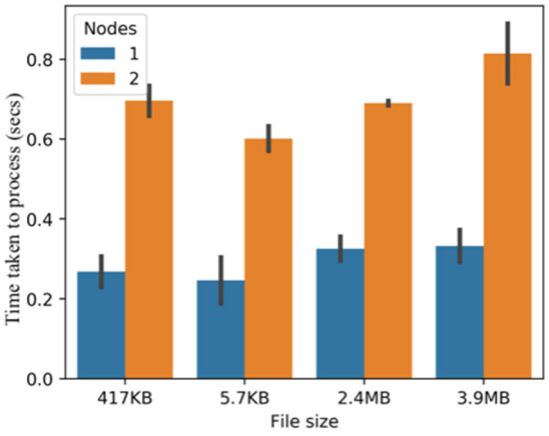

Figure 10: Wordcount on VM cluster using Blaze framework.

The following results were observed when the framework was used for *K*-means clustering program:

As seen from Figure 8, *K*-means performance was optimal and with increasing dimensions, the algorithm performed better. Scalability was displayed with increasing performance with nodes.

As seen in Figure 9, *K*-Means clustering on Blaze was tested to be faster than Spark implementation by a large margin. The scalability was close to linear and halved for each rise in number of nodes.

5.2 Wordcount using MR

The time taken for each experiment is measured against that number of points and number of nodes processed. It is observable that the framework tends to increase performance as number of nodes increases and displays linear scalability as observable in Figure 10.

Wordcount is the hello-world program of MR. This task was inefficient in terms of scalability as the framework tended to increase processing time with the increase in nodes. This has to be fixed in future work. Part of the issue of scalability can be addressed to the shuffle phase unable to facilitate movement of large loads of KV pairs which is unsuitable for low key ranges but on larger dataset, the scalability is linear as seen in Figure 11.

5.3 Pi estimation using MR

Pi estimation using Monte Carlo is an algorithm where random coordinates (x, y) are generated in mappers and if they fall within a certain range the mapper emits (key, 1), else emits (key, 0). The reducer sums over the key and estimates the value of pi using 4* (count of points inside/total count of points).

As seen in Figure 12, this algorithm, when implemented on the framework, was very efficient in terms of

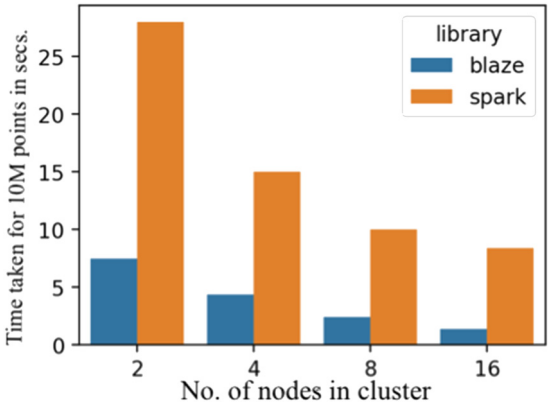

Figure 9: *K*-means clustering comparison between Blaze and Spark.

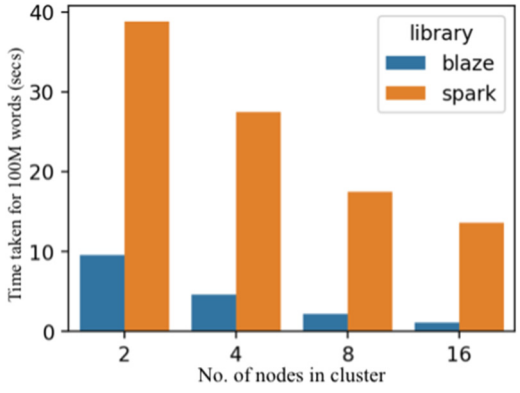

Figure 11: Wordcount comparison between Blaze and Spark.

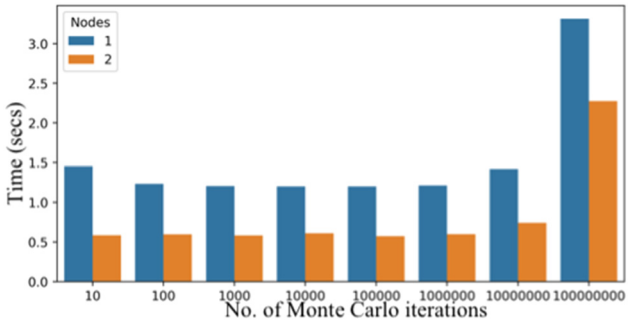

Figure 12: Pi estimation using Monte Carlo method on VM cluster using Blaze framework.

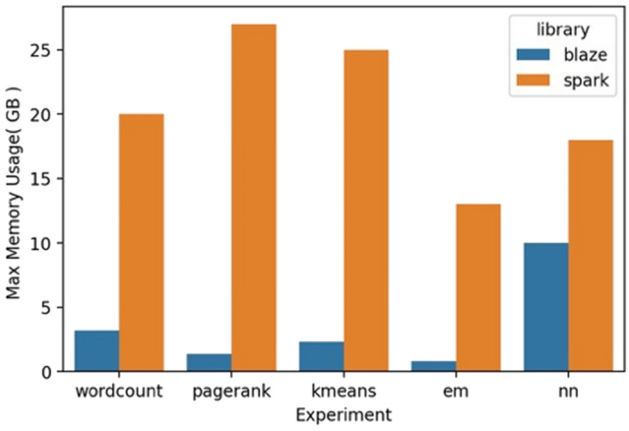

Figure 13: Memory usage difference between Blaze framework and Spark.

memory, speed, and scalability. The time taken for processing reduces almost linearly for increase in the number of nodes.

5.4 Peak memory usage comparison

The overall Memory usage was compared for different algorithms against Blaze and Spark to find differences in performance and efficiency and plotted in Figure 13.

6 Conclusion and future work

The entire C++-based framework for MR is very efficient and fast compared to a standard implementation. From a developer standpoint, it provides a simple and code-efficient implementation. From a deployment standpoint, it is clear that the MPI is not fault-tolerant, being one of the bottlenecks of the proposed system. The proposed architecture provides a feasible alternative to MR based on the JVM. It also provides an additional feature for Delayed reduction to provide an optimized and fully featured MR similar to Hadoop's implementation in terms of features but faster and easier to develop and deploy as a HPC-based alternative to the JVM-based Hadoop MR.

It also implements a few MR-based machine learning algorithms on the given framework. Furthermore, more algorithms can be implemented and tested in the future to prove the efficiency that HPC-based systems are far faster on compute-intensive tasks than JVM-based systems.

Acknowledgment: We would like to thank Vellore Institute of Technology, Chennai for providing the opportunity and support to work on this article.

Author contributions: All the authors of the papers contributed equally to all section making it difficult to separate the works hence we would like to not include an Authors Contribution section.

Conflict of interest: The authors declare that we do not have any financial interest (such as honoraria; educational grants; participation in speakers' bureaus; membership, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, knowledge, or beliefs) in the subject matter or materials discussed in this manuscript.

Data availability statement: The datasets generated during and/or analysed during the current study are available in the Open Sourced Corpus Repository [https://corpus.canterbury.ac.nz/purpose.html]. This was primarily used for WordCount algorithm. All other algorithms depend on Random Number Generation hence no particular dataset is needed.

References

[1] V. Kalavri and V. Vlassov, "MapReduce: Limitations, optimizations and open issues," *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2013. doi: 10.1109/trustcom.2013.126.

[2] "13 Big Limitations of Hadoop & Solution To Hadoop Drawbacks," By DataFlair Team. Accessed on: [Online], March 2019. https://data-flair.training/blogs/13-limitations-of-hadoop/.

[3] J.-F. Weets, M. K. Kakhani, and A. Kumar, "Limitations and challenges of HDFS and MapReduce," *2015 International Conference on Green Computing and Internet of Things* (*ICGCIoT*), 2015. doi: 10.1109/icgciot.2015.7380524.

[4] D. Jeffrey and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM51*, vol. 1, no. January 2008, pp. 107–13, 2008. doi: 10.1145/1327452.1327492.

[5] F. Chesani, A. Ciampolini, D. Loreti, and P. Mello, "MapReduce autoscaling over the cloud with process mining monitoring," *International Conference on Cloud Computing and Services Science 2016 Apr 23*, Cham, Springer, 2017, pp. 109–30. doi: 10.1007/978-3-319-62594-2_6.

[6] K. Chen, J. Powers, S. Guo, and F. Tian, "CRESP: Towards optimal resource provisioning for MapReduce computing in public clouds. parallel and distributed systems," *IEEE Tran. on*, vol. 25, pp. 1403–12, 2014. doi: 10.1109/TPDS.2013.297.

[7] V. Muthumanikandan and C. Valliyammai, "Link failure recovery using shortest path fast rerouting technique in SDN," *Wireless Pers. Commun.*, vol. 97, pp. 2475–95, 2017. doi: 10.1007/s11277-017-4618-0.

[8] S. Plimpton and K. Devine, "MapReduce in MPI for large-scale graph algorithms," *Parallel Comp.*, vol. 37, no. 610, pp. 632, 2011. doi: 10.1016/j.parco.2011.02.004.

[9] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, et al., "Mimir: memory-efficient and scalable MapReduce for large supercomputing systems," *2017 IEEE International Parallel and Distributed Processing Symposium* (*IPDPS*), 2017. doi: 10.1109/ipdps.2017.31.

[10] "Blaze: Simplified High Performance Cluster Computing by Junhao Li, Hang Zhang." arXiv:1902.01437 [cs.DC].

[11] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan, "MARIANE: MapReduce implementation adapted for HPC environments," *2011 IEEE/ACM 12th International Conference on Grid Computing, Lyon*, 2011, pp. 82–9.

[12] S. Kang, S. Lee, and K. M. Lee, "Performance comparison of OpenMP, MPI, and MapReduce in practical problems," *Adv. Multimedia*, vol. 2015, pp. 1–9, 2015. doi: 10.1155/2015/575687.

[13] "Microsoft Research. "[Online]. http://www.microsoft.com/windows azure/.

[14] Amazon, "Amazon Elastic Compute Cloud." [Online]. http://aws.amazon.com/ec2.

[15] V. Muthumanikandan, P. Singh, R. Chithreddy, "Cloud-based face and face mask detection system. In: *Intelligent Systems and Sustainable Computing. Smart Innovation, Systems and Technologies*, V. S. Reddy, V. K. Prasad, D. N. Mallikarjuna Rao, S. C. Satapathy, Eds, vol. 289. Springer, Singapore. 2022. doi: 10.1007/978-981-19-0011-2_38.

[16] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang, "Smart: A MapReduce-like framework for *in-situ* scientific analytics," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (*SC*), 2015.

[17] N. Hu, Z. Chen, Y. Du, and Y. Lu, "Mimir+: An optimized framework of MapReduce on heterogeneous high-performance computing system," *Network and Parallel Computing. NPC 2018. Lecture Notes in Computer Science*, F. Zhang, J. Zhai, M. Snir, H. Jin, H. Kasahara, M. Valero, (eds), vol. 11276, Cham, Springer, 2018.

[18] N. Nguyen and D. Bein, "Distributed MPI cluster with Docker Swarm mode," *2017 IEEE 7th Annual Computing and Communication Workshop and Conference* (*CCWC*), *Las Vegas, NV, USA*, 2017, pp. 1–7.

[19] K. Doucet and J. Zhang, "The creation of a low-cost Raspberry pi cluster for teaching," *WCCCE '19: Proceedings of the Western Canadian Conference on Computing Education*, 2019, pp. 1–5. doi: 10.1145/3314994.3325088.

[20] W. Zhao, H. Ma, and Q. He, Parallel k-means clustering based on MapReduce, *Cloud Computing. CloudCom 2009. Lecture Notes in Computer Science*, M. G. Jaatun, G. Zhao, C. Rong, (eds), vol. 5931, Berlin, Heidelberg, Springer, 2009.

[21] Z. Fadika and M. Govindaraju, "Delma: Dynamically elastic mapreduce framework for CPU-intensive applications," *CCGRID*, 2011, pp. 454–63.

[22] "Alpine Linux with MPICH for developing and deploying distributed MPI programs," Accessed on: Jan, 2020 [*Online*]. https://hub.docker.com/r/nlknguyen/alpine-mpich.

[23] "High Performance MapReduce-First Cluster Computing," Junhao12131. Accessed on: Feb 2020. https://github.com/junhao12131/blaze.