A Hybrid Spark MPI OpenACC System

Waleed Al Shehri[1]*, Maher Khemakhem[2], Abdullah Basuhail[3] and Fathy E. Eassa[4]

Department of Computer Science, King Abdul-Aziz University, Jeddah, KSA

Summary
Apache Spark is a common big data platform that is built based on a Resilient Distributed Dataset (RDD). This data structure abstraction is able to handle large datasets by partitioning and computing the data in parallel across many nodes. In addition, Apache Spark also features fault tolerance and interoperability with the Hadoop ecosystem. However, Apache Spark is written in high-level programming languages which do not support high parallelism like other native parallel programming models such as Message Passing Interface (MPI) and OpenACC. Furthermore, the use of the Java Virtual Machine (JVM) in the Spark implementation negatively affects performance. On the other hand, the tremendous volume of big data may not be suitable for distributed tools such as MPI and OpenACC to support a high level of parallelism. The distributed architecture of big data platforms is different from the architecture of High Performance Computing (HPC) clusters. Big data applications running on HPC clusters cannot exploit the capabilities afforded by HPC. In this paper, a hybrid approach is proposed that takes the best of both worlds by handling big data with Spark combined with the fast processing of MPI. In addition, the availability of graphics processing units (GPUs) available in modern systems can further speed up the computation time of an application. Therefore, the hybrid Spark+MPI approach may be extended by using OpenACC to include the GPU processor as well. To test the approach, the PageRank algorithm was implemented using all three methods: Spark, Spark+MPI and Spark+MPI+OpenACC.

Key words:
High-Performance Computing; Big Data; Spark; MPI; OpenACC; Hybrid Programming model; Power Consumption;

1. Introduction

The concept of big data emanated from the dramatic growth in the amount of data produced in scientific and commercial fields. As a result, many big data technologies have emerged to address the challenges of collecting, analyzing, and storing such enormous amounts of data. It is noticeable that high-performance computing (HPC) and big data applications are converging due to the capabilities of each paradigm. Big data technologies are able to handle enormous datasets, while HPC is concerned with performing computations as fast as possible by integrating heterogeneous hardware and crafting software to exploit high levels of parallelism [1].
Apache Spark is a popular distributed cluster computing framework for big data analytics. Spark's basis is in a Resilient Distributed Dataset (RDD), which is a special data structure abstraction that can be partitioned and computed in parallel across many compute nodes.
Despite some attractive features of Spark, such as interoperability with the Hadoop ecosystem and fault tolerance, computation is slower than it could be since it is implemented using the Java Virtual Machine (JVM). In contrast, other distributed tools such as Message Passing Interface (MPI), which is natively implemented, outperform Spark [2], For example, a case study on large matrix factorizations found that a C implementation with MPI is 4.6–10.2 times faster than Spark on an HPC cluster with 100 compute nodes [3].
Alternatively, the large volume of big data may hinder parallel programming models such as MPI, Open Multi-Processing (OpenMP) and accelerator models (CUDA, OpenACC, OpenCL) from supporting high levels of parallelism [4].
In addition, the architecture of big data platforms is distributed, which differs from the architecture of HPC clusters [5].
Furthermore, resource allocation in a HPC environment is an arduous job due to differences in the software stack of both HPC and big data paradigms [6].
As a result, there is usually a performance gap when running big data applications on HPC clusters. In this paper, a hybrid approach is proposed that takes the best of both worlds by handling big data with Spark combined with fast processing of MPI. In addition, the availability of GPUs in modern systems can further speed up computation time of an application. Therefore, the hybrid Spark+MPI approach suggested above may be extended by using OpenACC to include the GPU processor as well.
To test the approach, the PageRank algorithm is implemented in all three methods, Spark, Spark+MPI, and Spark+MPI+OpenACC.
For this work, Spark was used together with MPI and OpenACC. MPI and OpenACC share a native implementation base while Spark is implemented using the JVM, which operates in higher domain. In order to bridge the two environments, we had to setup individual development environments. Further, in order to develop OpenACC applications, we used the PGI compiler. This gave us the benefit of an Open MPI implementation that was compiled with the PGI compiler and recognized the OpenACC directives however would only support Nvidia GPUs.

Manuscript received May 5, 2019
Manuscript revised May 20, 2019

Spark applications were developed in Scala using IntelliJ IDEA. For MPI and OpenACC code was written in C using Microsoft Visual Code.
The remaining sections of this paper are structured as follows: Section 2 highlights the related work to our approach. The experimental method is discussed is section 3 followed by results and, finally, a conclusion.

2. Related Work

It was observed that the MPI-based HPC model shows that Spark or Hadoop-based large data model is order of magnitude for verity of different applications such as large-scale matrix factorization [3], graph analytics [8], [9], k means [10], support vector machine and k-nearest neighbor's [11]. A recently analysis based on Spark showed that compute load is the primary bottleneck in a large number of applications like deserialization time and specifically serialization [12]. Lu et al. [13] conclude that replacing map with an MPI derivative reduced communication and leads to better performance. A drawback of this research was that it was not a drop-in replacement for Hadoop and there was a need to recode it to use the Data MPI. Here it is shown that MPI applications are extendable and so become elastic for a number of nodes through periodic data by redistribution among the MPI ranks [14]. We use MPI as a programming framework since similar benefit cannot be obtained by processing with cloud Spark or Hadoop such as fault tolerance and high productivity. Fagg, Dongarra proposed FTMP [15], an effort to add fault tolerance to MPI from the year 2000. Fault tolerance mechanisms in the MPI standard are still not integrated so proposed solutions continue to put forth such as Fenix [16]. Fenix and Spark have a large productivity gap between them. SWAT [17] is limited to single node optimizations, it cannot benefit from communication improvements provided by MPI. Thrill [18] is a project building a Spark data processing system that uses C++ and MPI for communication.
Alchemist [19] interfaces between MPI libraries and Spark and observes that such interfacing speeds up linear algebra routines. Improved performance comes from the comparative overhead of moving data over the network between Spark nodes and Alchemist and there is still benefit working in the Spark environment.
Smart MLlib [20] enables Spark to call custom machine learning code implemented in C++ and MPI. The system appears to no longer be under active development and has not been shown to be scalable.
Based on the previous related work, and to the best of our knowledge, there is no contribution yet that integrates the three programming models Spark, MPI, and OpenACC. Such integration can take the best of each world to accelerate big data applications, particularly in HPC environments where pure big data applications are not able to exploit HPC capabilities.

3. Experimental Method

The proposed system is implemented on a single system machine with the following specifications (Table 1):

Table 1: Target Machine Specifications

OS	Ubuntu 18.04.1
Memory	description: System Memory physical id: 3b slot: System board or motherboard size: 16GiB
Cache	*-cache:0 description: L1 cache size: 384KiB *-cache:1 description: L2 cache size: 1536KiB *-cache:2 description: L3 cache size: 9MiB
CPU	description: CPU – 12 Cores product: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz

The physical topology of the target machine can be seen as Figure 1.

3.1 RDD-based approach

Spark's core is RDD. Its distributed nature, one of its key features, sees data stored in partitions and distributed across nodes. Each partition is a unit of parallelism and consists of records that contain a subset of the data being processed. Spark creates partitions and divides data among them based on the following parameters:

- Number of available cores
- Available memory

The partitioning process can be customized. Processing begins once data is partitioned into subsets. Our approach takes over from there. Spark generally performs the following operations on data:

- Transformations
- Actions

In this approach data is passed for each partition to a worker that is responsible for the execution of the algorithm. Each partition's data can be computed independently and the individual results can be combined to produce the overall result. This is an inherently parallel solution.

3.2 The Worker

The worker is responsible for performing the computations of the algorithm being evaluated. It receives data, performs computations, and then returns the results to the Spark

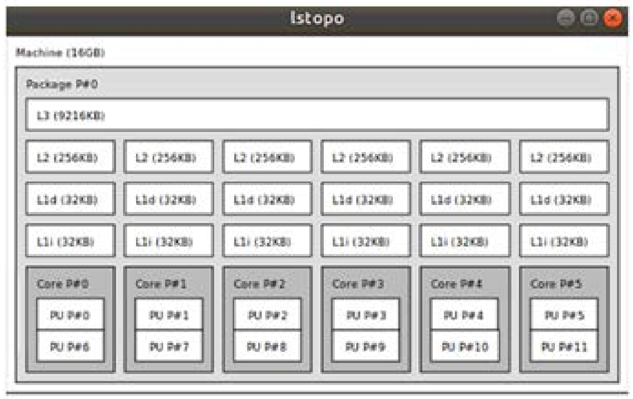

Fig. 1 Physical Topology for the target machine

environment, where all the worker sub-results are combined to get the final result.
The worker needs to be implemented by employing parallel programming models such as MPI and OpenACC. Such native implementations support more parallelism than pure Spark, which is a high-level programming model.

3.2.1 MPI

Message passing interface (MPI) is a standard for message passing, used for parallel computing. The two main implementations of the MPI standard are MPICH and Open MPI. For this approach, Open MPI is used with code compiled using the PGI compiler.
The worker is implemented in C and uses the MPI libraries to send and receive data. Point-to-point communication is used in a circular fashion (starts with process 0 and ends with process 0).

3.2.2 OpenACC

OpenACC can be used if a GPU is available in any of the HPC cluster nodes. This can give an extra speed boost to the processing time of an algorithm. The PGI compiler was required to use OpenACC. MPI in combination with OpenACC was used so that a cluster solution can be employed. If certain nodes do not have a GPU available, the worker needs to have an alternate solution that does not require a GPU.
GPU’s have limited memory and only basic operations are available on the processor; any additional method definitions along with the data is required to be transferred onto the GPU memory. This creates an additional task for the programmer to break down the code to elemental form, which could be difficult for complex algorithms.

3.3 PageRank Algorithm in Spark

The PageRank algorithm is based on the concept of weighted graphs and is the backbone of the Google search engine. It calculates the probability distribution of a link being clicked and a page being visited. In short, more references/links a page has from other pages, the greater the probability that page will be visited. The algorithm is defined by the following equation:

$$PR^{t+1}(v) = r + (1-r) \times \sum_{u|(u|v)\in E} \frac{PR^{t}(u)}{degree\,(u)}$$

Equation 1

As can be observed from the equation, the PageRank implementation can be parallelized as per the requirement of the approach and sub results can be combined to reach a final result; fulfilling the requirement for qualification for the proposed approach.
The PageRank algorithm is used as the basis for comparing the three methods described above. The pure Spark implementation makes use of Spark’s built-in PageRank method to calculate ranks for provided input.

3.3.1 Dataset

The data set used for experimentation consisted of Twitter circles. The data were crawled from public sources and represent Twitter user connections that are masked numerically [7]. The data consist of 2,420,766 edges. Each edge represents a connection between two nodes, where a node represents a Twitter user in a circle.
Applying the PageRank algorithm to this dataset calculates the highest ranked Twitter user. For the sake of experimentation, we spliced the data into a number of partitions. The Twitter data was split by subtracting 25 % of the 2,420,766 edges to create the first split and consequently each split was produced by reducing 10 % of the data from the first split. The process continued until the size was reduced to 60 % of the original data size as any further splitting would make the data set too small.
The full dataset was too large for the new system to handle. Data shuffling was barred and single system resources were used. The system crashed and the pure Spark implementation took an infinite amount of time which led to the conclusion that additional steps were required to handle data beyond a certain size. However, the results on datasets the system could handle clearly showed excellent improvements in processing time as compared to the pure Spark implementation.

3.4 Spark with MPI

To integrate Spark with MPI (Figure 2), the glom method with Spark is used, which allows data to be accessed as a traversal object. The biggest advantage of this method is that data is neither shuffled nor relocated among partitions, which is the root cause of major delays in Spark executions. For each partition, as explained in the previous section, a worker implemented in C using MPI libraries is called, and Spark is used for the PageRank algorithm.

```
var PR = Array.fill(n)( 1.0 )
 val oldPR = Array.fill(n)( 1.0 )
 for( iter <- 0 until numIter ) {
   swap(oldPR, PR)
   for( i <- 0 until n ) {
     PR[i] = alpha + (1 - alpha) * inNbrs[i].map(j => oldPR[j] / outDeg[j]).sum
   }
 }
```

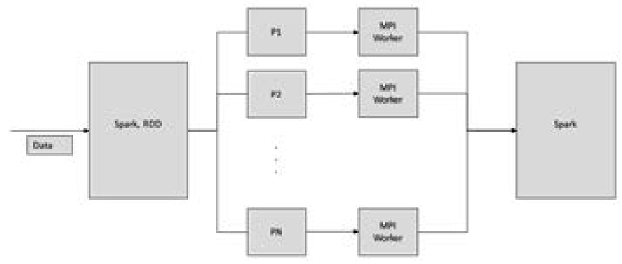

Fig. 2 A dual programming model of Spark and MPI

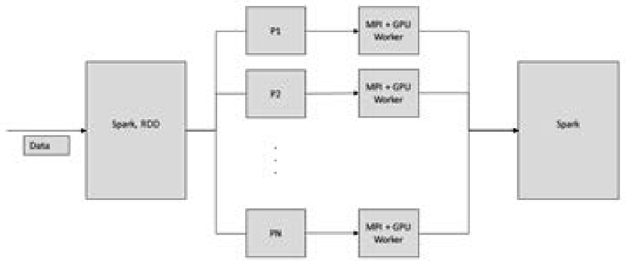

Fig. 3 A hybrid programming model of Spark, MPI and OpenACC

In this case, the worker receives a subset of edges and returns the ranks of the edges within those subsets. Here, edges represent links; the job of the worker is to:

a) Determine the number of vertices (also referred as nodes or links).
b) Determine the degree of each vertex.
c) Using information from (a) and (b) to distribute data between MPI processes to calculate the page ranks.

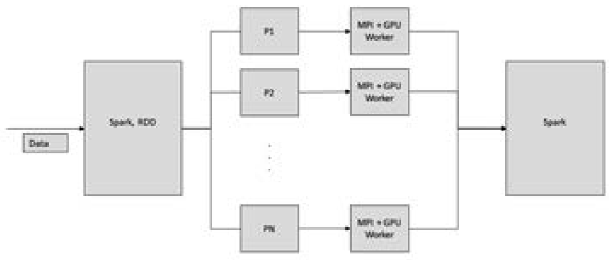

Fig. 3 A hybrid programming model of Spark, MPI and OpenACC

3.4 Spark with MPI, with OpenACC support

This implementation (Figure 3) is an extension of Spark with MPI. As explained in the previous section, the code needs to check for the availability of a GPU on a node and assign the relevant code, otherwise it must be able to run the simple MPI code. The logic of the algorithm is similar to that explained in the Spark with MPI section above, but it is broken down to smaller steps in the implementation.

4. Results and Discussion

The comparison of the three methods showed that Spark+MPI+OpenACC gave the best processing time against various input data sizes (**Figure 4**).

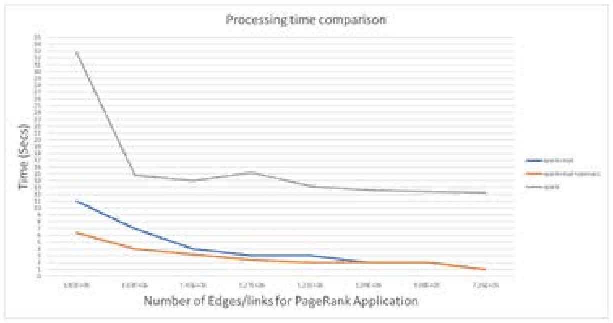

Fig. 4 Processing Time Comparison

An anomaly was observed for Spark in the middle of the graph. This can be associated to a type of data shuffling that resulted for that dataset.
The most overhead in Spark is generated when the dataset increases and, for the sake of computation, data movement

is required. In our scheme we have limited any data movement.

For small datasets, Spark gives better performance although the proposed schemes may be optimized to surpass Spark in that domain as well.

This Produces Optimal results though it requires a custom scheme for resource management and load balancing so that MPI assigns nodes based on the locality of the data, which will be addressed in future work.

In addition, data is currently being passed as a shared copy, which produces a memory load on the entire system. This is a problem that needs to be tackled. A suggestion is to send a reference to the original data. However, data integrity and failsafe needs to be ensured.

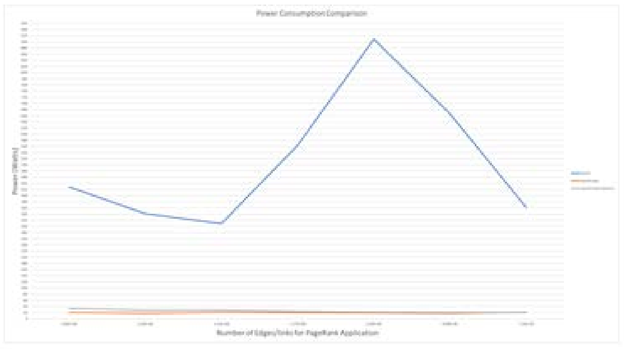

Fig. 5 Power Consumption Comparison

(Figure 5) above shows power consumption comparison between Spark, Spark+MPI and Spark+MPI+OpenACC at 512 memory for executor and driver. Spark's power consumption is non-linear and for certain datasets heavy power consumption is observed which can be attributed to data shuffling and clearing of accumulators.

5. Conclusion

This paper proposed a Hybrid Spark MPI OpenACC system, which is an integration of these three programming models, to enhance Spark-based big data applications. The benefit of such integration is anticipated to be significantly enhanced performance without sacrificing the power consumption metric. This can be seen clearly in HPC cluster environments where Spark and similar platforms, that are written in high level programming languages, fail to exploit HPC capabilities and support high parallelism. As a result of the approach described in this paper, performance was enhanced although it is on a single system. For future work, this test will be implemented in a real HPC cluster environment with the need to consider data locality to reduce data movement. This can be achieved by employing resource management techniques that will extract the physical topology and map the virtual topology of a big data application in a way that exploits HPC resources effectively and achieve data locality. This is predicted to more reduction in power consumption along with enhancing performance.

References

[1] D. A. Reed and J. Dongarra, "Exascale computing and big data," Commun. ACM, vol. 58, no. 7, pp. 56–68, 2015.

[2] M. Anderson et al., "Bridging the gap between HPC and big data frameworks," Proc. VLDB Endow., vol. 10, no. 8, pp. 901–912, 2017.

[3] A. Gittens et al., "Matrix factorizations at scale: A comparison of scientific data analytics in spark and C+MPI using three case studies," in 2016 IEEE International Conference on Big Data (Big Data), 2016, pp. 204–213.

[4] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," Mob. Networks Appl., vol. 19, no. 2, pp. 171–209, 2014.

[5] P. Xuan, J. Denton, P. K. Srimani, R. Ge, and F. Luo, "Big data analytics on traditional HPC infrastructure using two-level storage," Proc. 2015 Int. Work. Data-Intensive Scalable Comput. Syst. - DISCS '15, pp. 1–8, 2015.

[6] H. R. Asaadi, D. Khaldi, and B. Chapman, "A comparative survey of the HPC and big data paradigms: Analysis and experiments," Proc. - IEEE Int. Conf. Clust. Comput. ICCC, pp. 423–432, 2016.

[7] "SNAP: Network datasets: Social circles." [Online]. Available: https://snap.stanford.edu/data/ego-Twitter.html. [Accessed: 28-Apr-2019].

[8] N. Satish et al., "Navigating the maze of graph analytics frameworks using massive graph datasets," in Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14, 2014, pp. 979–990.

[9] G. M. Slota, S. Rajamanickam, and K. Madduri, "A Case Study of Complex Graph Analysis in Distributed Memory: Implementation and Optimization," in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 293–302.

[10] S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. C. Fox, "A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures," in 2014 IEEE International Congress on Big Data, 2014, pp. 645–652.

[11] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf," Procedia Comput. Sci., vol. 53, pp. 121–130, Jan. 2015.

[12] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making Sense of Performance in Data Analytics Frameworks," 12th USENIX Symp. Networked Syst. Des. Implmentation (NSDI 2015), pp. 293–307, 2015.

[13] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "DataMPI: Extending MPI to Hadoop-Like Big Data Computing," in 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 829–838.

[14] A. Raveendran, T. Bicer, and G. Agrawal, "A Framework for Elastic Execution of Existing MPI Programs," in 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011, pp. 940–947.

[15] G. E. Fagg and J. J. Dongarra, “FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World,” Springer, Berlin, Heidelberg, 2000, pp. 346–353.

[16] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, “Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales,” in SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 895–906.

[17] M. Grossman and V. Sarkar, “SWAT,” in Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing - HPDC ’16, 2016, pp. 81–92.

[18] T. Bingmann et al., “Thrill: High-performance algorithmic distributed batch data processing with C++,” in 2016 IEEE International Conference on Big Data (Big Data), 2016, pp. 172–183.

[19] A. Gittens et al., “Accelerating Large-Scale Data Analysis by Offloading to High-Performance Computing Libraries using Alchemist,” in Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining - KDD ’18, 2018, pp. 293–301.

[20] D. Siegal, J. Guo, and G. Agrawal, “Smart-MLlib: A High-Performance Machine-Learning Library,” in 2016 IEEE International Conference on Cluster Computing (CLUSTER), 2016, pp. 336–345.