Journal of Computer Sciences

Original Research Paper

Novel Apache Spark based Algorithm to Solve Dirichlet Problem for Poisson Equation in 3D Computational Domain

Shomanov Aday and Mansurova Madina

Department of Computer Science, al-Farabi Kazakh National University, Almaty, Kazakhstan

Article history
Received: 09-08-2016
Revised: 04-10-2016
Accepted: 14-12-2016

Corresponding Author:
Shomanov Aday
Department of Computer Science, al-Farabi Kazakh National University, Almaty, Kazakhstan
Email: adai.shomanov@gmail.com

Abstract: Parallel computations are essential tool in solving large-scale computationally demanding problems. Due to large diversity and heterogeneity of the currently available parallel processing techniques and paradigms it is usually difficult to find the right solution that will perform well according to every performance metric. As one of the recent developments in parallel computing Apache Spark framework allows to process petabyte-scale data and possesses properties such as fault tolerance, scalability, load balancing and mechanisms of in memory computations across nodes of the cluster. All of these features are attractive for high performance scientific computing. It has been shown that Apache Spark outperforms Hadoop implementation of some machine learning algorithms by orders of magnitude. Since Hadoop platform is not well suited for iterative computing, typical for many computational problems, in this study we investigate performance characteristics of Apache Spark on scientific computing problems, particularly for solving Dirichlet problem for Poisson's equation. An algorithm for solving Dirichlet problem for Poisson's equation is described and analyzed and compared to optimized Hadoop-based implementations. Apache Spark uses new distributed data structure called RDD. Presented algorithm consists of operations on RDD such as mapping, grouping and partitioning. The benefits and drawbacks of the algorithm as well as applicability for stencil type computations are discussed and analyzed.

Keywords: Hadoop, Spark, RDD, HPC

Introduction

In a modern world there are a lot of large-scale and computationally intensive problems that require highly efficient and well designed approaches to solve them. Historically, large-scale computational problems have been solved by means of HPC clusters using MPI paradigm where the main complexity was to design computational domain, perform the most efficient domain decomposition schemes and algorithms and find cost-effective and feasible hardware solutions. MPI has some drawbacks such as idle CPU usage while exchanging data between nodes, reliability and data loss due to node failures or network collapses. Hadoop is a distributed computing platform that uses MapReduce paradigm applied to data stored in Hadoop Distributed File System (HDFS). Hadoop possesses some advantages such as fault tolerance, scalability and load balancing. Node failures in Hadoop do not necessarily lead to termination of the program and in cases when data on the failed nodes had been previously properly replicated computation can be continued without any complications. Hadoop, on the other hand, is not designed to solve iterative problems efficiently and in general lacks data exchange mechanisms between nodes.

We, therefore, designed and implemented our novel approach using Apache Spark with the aim to resolve some of the above mentioned drawbacks of using MPI only or Hadoop only approaches for large-scale scientific computational problems.

Apache Spark is a framework for large-scale data processing with the following main features (Zaharia *et al*., 2010):

Data abstractions called Resilient Distributed Datasets (RDD), which allow to perform bulk operations on the data in parallel and cache intermediate results in memory. Each RDD consists of several data blocks that are divided across cluster nodes (Fig. 1). Operations on RDD can be of two types: Transformations and actions (Fig. 2).

Transformations in Spark are performed lazily, i.e., application of transformations is delayed until some action on RDD is performed. Whereby, Spark can optimize execution of transformations, for example, by rearranging the order in which they are applied.

In memory caching of RDD data is performed in order to optimize the speed of data access operations later in computation process**.** Caching is performed on each partition of RDD. Since partitions may reside on different nodes of the cluster caching is performed separately on each node.

RDD offers the following set of main transformations: Map, filter, flatMap, groupByKey, union, join, crossProduct. The following are the main actions performed on RDD: Count, collect, reduce, save.

From Fig. 1 it can be seen that the data blocks from the same RDD might be placed on different cluster nodes. To forcefully put data from one partition of RDD to single physical node there exist special partitioning tools provided by Spark. Consequently, by using partitioning tools data placement in a cluster can be controlled according to specific purposes of the problem.

Besides the ability to persist data in memory such that later operations could be performed sufficiently fast RDD also exploits data locality property. Data locality property implies that each task should perform operations on those partitions of RDD which are located on its own local memory or which can be fetched from other nodes with minimal network workload and computational resources used.

Hadoop has poor performance on iterative tasks since each iteration results in the loss of the job`s execution context and consequently necessary data for the next iteration should be loaded again in memory from HDFS. Contrary to that RDD keeps as much necessary data in memory as possible.

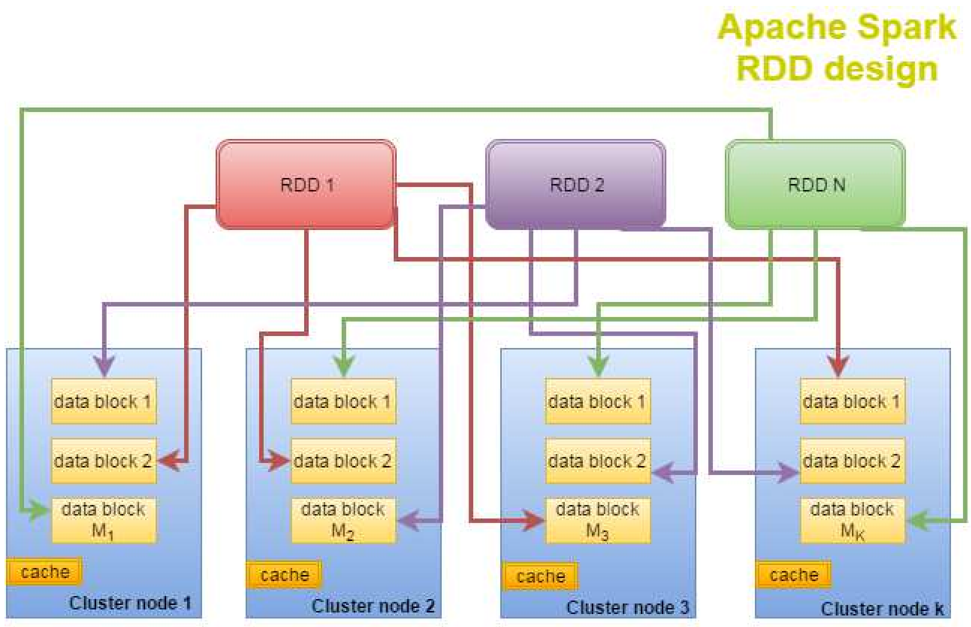

Fig. 1. Apache spark RDD design

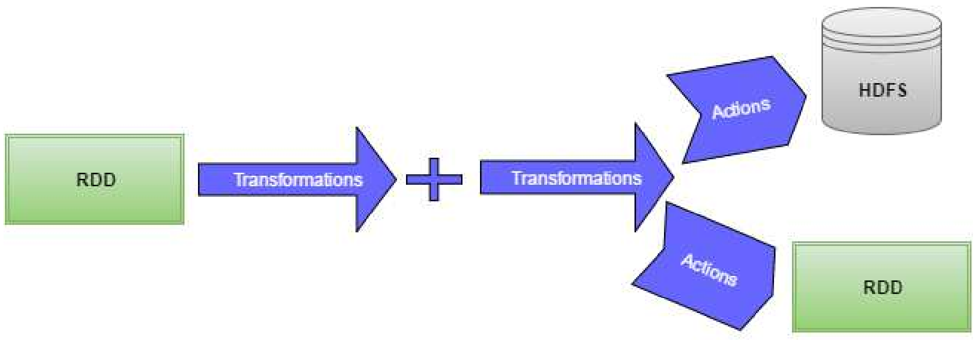

Fig. 2. Spark workflow for single operation list

Figure 2 shows the general scheme how Spark performs operations on RDD. There is some set of transformations followed by actions. Actions result either in creating new RDD or dumping data into HDFS file system.

Fault-tolerance in RDDs is achieved through keeping lineage i.e. set of particular transformations that have led to the current state of RDD. If some data is lost due to node failure, transformations can be performed again in that particular set of nodes that kept replica of the data stored in the failed node and after that RDD will be restored back to its current state.

We present in our work an iterative Apache Spark solution to the Dirichlet problem for Poisson's equation on three-dimensional computational domain which allows efficient iterative execution and additionally provides caching of locally kept chunks of data.

Related Work

There is large scope of problems in different areas of science that have already been successfully solved by using Apache Spark (Freeman, 2014; Horlacher *et al*., 2014; Zhao *et al*., 2015). Apache Spark development initially was motivated by low performance of machine learning tools for large-scale data processing available at the time. Now Spark offers much broader range of techniques to tackle different kinds of problems.

There are currently not so many alternatives to choose from to replace or augment MPI computational paradigm for large-scale scientific problems with iterative schemes and research is ongoing with varied success in this important field.

The obvious advantage of MPI over Apache Spark is that MPI potentially could be broadened to wide range of applications in HPC and still be sufficiently fast. However, the main issue with MPI is its lack of built-in failure resistance. Failures could be problematic for long-running jobs in setting of large number of computing nodes. There are different approaches to avoid failures in MPI environment but they are tedious and error-prone to implement. Thorough treatment of these approaches are given in (Gropp and Lusk, 2004).

Apache Spark design essentially can be treated as a generalization of Mapreduce programming paradigm in the context of distributed programming models. Mapreduce can be viewed as a series of parallel map tasks followed by series of parallel reduce tasks. Functionality of the map is to derive key/value pairs from raw input according to some criteria. Reduce on the other hand takes list of values with specified key as an input and outputs different set of key/value pairs generated from the input list. Spark offers aside from map and reduce several other operations mentioned earlier and in general abstracts away these operations into transformation concept.

Many works are devoted to improving speed of running Mapreduce based programs. In (Li *et al*., 2016) authors describe several avenues for improvement in Hadoop Mapreduce framework:

- Developing more efficient job scheduling mechanism that takes into account non homogeneous distribution of resources in a distributed system
- Improving programming model by developing advanced iterative processing routines that would allow more efficient job execution
- Developing more convenient real-time processing by improving streaming functionality
- Extending the capabilities of the system by allowing parallel execution of map and reduce tasks

Apache Spark is believed to show better performance according to second and third items shown above.

Apache Spark is based on RDD distributed data structure storage. In (Zaharia *et al*., 2012) authors described RDD internal design and properties and demonstrated its ability to perform in-memory computations on large clusters in a fault-tolerant way. In the paper authors also reported large speed-up on iterative graph and machine learning algorithms by using Apache Spark over PGAS and Hadoop. Considering conceptual differences of global-memory access languages such as PGAS and other parallel programming languages with different memory abstractions there is a trade-off between maintaining granularity of elements in memory and performing bulk operations on these elements. The main advantage over PGAS model is that RDD operations are coarse-grained therefore reducing overhead of storing states of each element in a distributed environment.

In (Lu *et al*., 2014; Lu and Liang, 2016) authors presented a new high-performance communication library based on MPI communication primitives called DataMPI. As a result they showed that using DataMPI communication primitives one could achieve performance gain of up to 32% compared to Hadoop communication primitives. Authors also generalize communication patterns into 4D bipartite communication model and key-value communication model, which fits into the requirements of Hadoop-like system specifications and could potentially lead to better design of communication sub-systems in Big Data frameworks.

In (Reyes-Ortiz *et al*., 2015) authors compare Apache Spark performance with MPI/OpenMP based on KNN and Pegasos SVM machine learning algorithms. The results showed that MPI/OpenMP approach is still more than 10 times faster in terms of running time, however, one should note that Spark has an advantage of caching and authors did not mention this in their paper.

In (Lu *et al*., 2011) authors describe hybrid framework of using MPI as a pipeline to exchange an intermediate data between concurrently running reduce and map processes. The resulting solution outperforms some of the Hadoop or MPI-Mapreduce implementations on three applications: WordCount, Distributed Inverted Indexing and Distributed Approximate Similarity Search.

Parallel Algorithm for Solving the Dirichlet Problem for Poisson's Equation using Hadoop Spark

Dirichlet problem arises in many areas of physics such as fluid dynamics, electromagnetism and gravity due to its ability to describe the behavior of fluid, electric, gravitational potentials. Exact analytical solutions for Dirichlet problem is only limited by specific cases in appropriate domains therefore in majority of situations numerical approaches to find solution to the problem is applied.

3D model of Dirichlet problem for Poisson's equation in a hypercube domain $D = \{(x,y,z): 0\le x\le l_1, 0\le y\le l_2, 0\le z\le l_3\}$ can be described by the following set of equations:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}=f(x,y,z)$$
$$u_\Gamma=\varphi(x,y,z)$$

Let define number of mesh points in x, y and z coordinates to be equal to N_1, N_2 and N_3. Then we could obtain computational mesh with step size $\Delta x=\frac{l_1}{N_1}$ in x direction, $\Delta y=\frac{l_2}{N_2}$ in y direction and $\Delta z=\frac{l_3}{N_3}$ in z direction.

Performing discretization using finite-difference method we obtain the following explicit iterative scheme:

$$u_{i,j,k}^{n+1} = \left(\begin{array}{l}\dfrac{u_{i+1,j,k}^n+u_{i-1,j,k}^n}{\Delta x^2}+\dfrac{u_{i,j+1,k}^n+u_{i,j-1,k}^n}{\Delta y^2}\\ +\dfrac{u_{i,j,k+1}^n+u_{i,j,k-1}^n}{\Delta z^2}-f_{i,j,k}\end{array}\right)\Bigg/\left(\frac{2}{\Delta x^2}+\frac{2}{\Delta y^2}+\frac{2}{\Delta z^2}\right)$$

where, $u_{i,j,k}$ and $f_{i,j,k}$ -values of u and f functions in (i,j,k) point of computational mesh.

Here we briefly overview the general idea behind Hadoop-based algorithm for solving Dirichlet problem for Poisson equation proposed by the authors in different paper (Mansurova *et al*., 2014).

Hadoop based implementation for solving Dirichlet problem has the following general structure:

Step 1. In Map phase algorithm maps each point in computational domain to certain reducer that will perform computations in its own sub-domain.
Step 2. In Reducer phase each point is mapped inside three-dimensional array to perform stencil computations according to Poisson`s equation difference scheme. The computed new internal points in current iteration are stored to local file system to avoid redistribution of them across the nodes of the cluster during the next iteration and new boundary points are reduced and written to output HDFS directory.
Step 3. If iteration count is reached the limit program will terminate, otherwise algorithm will continue with step 1.

Apache Spark algorithm for solving Dirichlet problem for Poisson equation uses object based representation of points in the computational domain. Point class is used for storing single point in computational mesh. Every point consists of the following properties: Integer partition_id, integer number x for x coordinate, integer number y for y coordinate, integer number z for z coordinate, floating-point number value for value in specific point (x,y,z) of the discrete mesh. Partition property is responsible for storing partition number of the point that is sub-domain identifier.

Algorithm general scheme is depicted in the Fig. 3. Algorithm consists of several steps each step performs certain operations on RDDs. For the algorithm we have chosen 1D domain decomposition method.

1D decomposition method tells that domain should be divided by x coordinate into contiguous sub-arrays.

Number of sub-arrays should be specified beforehand in a driver program. Since we need to perform stencil computations for every point in computational domain we have to specify procedure by which to perform exchange of boundary values between neighboring sub-domains in the end of the each iteration.

Stepwise description of the algorithm is described by the following scheme:

Step 1. Generate initial computational domain and store points in RDD<Point>
Step 2. Map each entry of RDD<Point> to PairRDD<key,Point>, where key represent partition number and the value is a Point.
Step 3. Perform grouping operation such that points are divided into several partitions where operations on each partition are performed in parallel

Step 4. Perform mapping transformation on each partition in a following way:

- Create 3-dimensional array for every partition and map values of points of that partition to corresponding elements of that array
- Perform computation on partitions based on obtained Poisson's equation's difference scheme
- Map back elements of 3-dimensional array to list of points
- Return as a result of map operation list of newly computed values of points stored in RDD<Point>

Step 5. Check for termination condition:

- If termination condition is met go to step 6
- If termination condition is not met go to step 2

Step 6. Store resulting values of points in HDFS (Hadoop distributed file system)

Important to note that before performing any actions or transformations on PairRDD<key,Point> we have done additional partitioning in order to specify that each partition should be located on separate cluster node. This is necessary optimization in order to avoid huge network resources utilization and therefore very poor performance of the program.

The slowest part of the algorithm is generating initial computational domain since it cannot be performed in parallel due to design of Spark`s RDD storage abstractions. This operation is performed in 3 steps:

- Generate points sequentially on master node
- Write points to HDFS
- Extract points from HDFS to RDD

Since each partition of RDD is defined to store only points with only one certain key, according to algorithm there must exist mechanism to exchange boundary points between different partitions. The main bottleneck in this algorithm is additional overhead associated with performing that exchange. In order to perform exchange operation we need to apply mapping, grouping and partitioning transformations in each iteration of the algorithm. Mapping transformation will map Point elements into tuple of Point and partition number. Partition number is identified by Point coordinates. After that Points are grouped together by applying groupByKey transformation and finally every partition is separated into locations by applying partitionBy transformation. Such operations slightly degrade performance of the program leading to moderate speed-up.

The major difference between Hadoop based implementation and Spark implementation is that Spark solution uses in-memory computations and thus is suited better for iterative tasks such as Dirichlet problem.

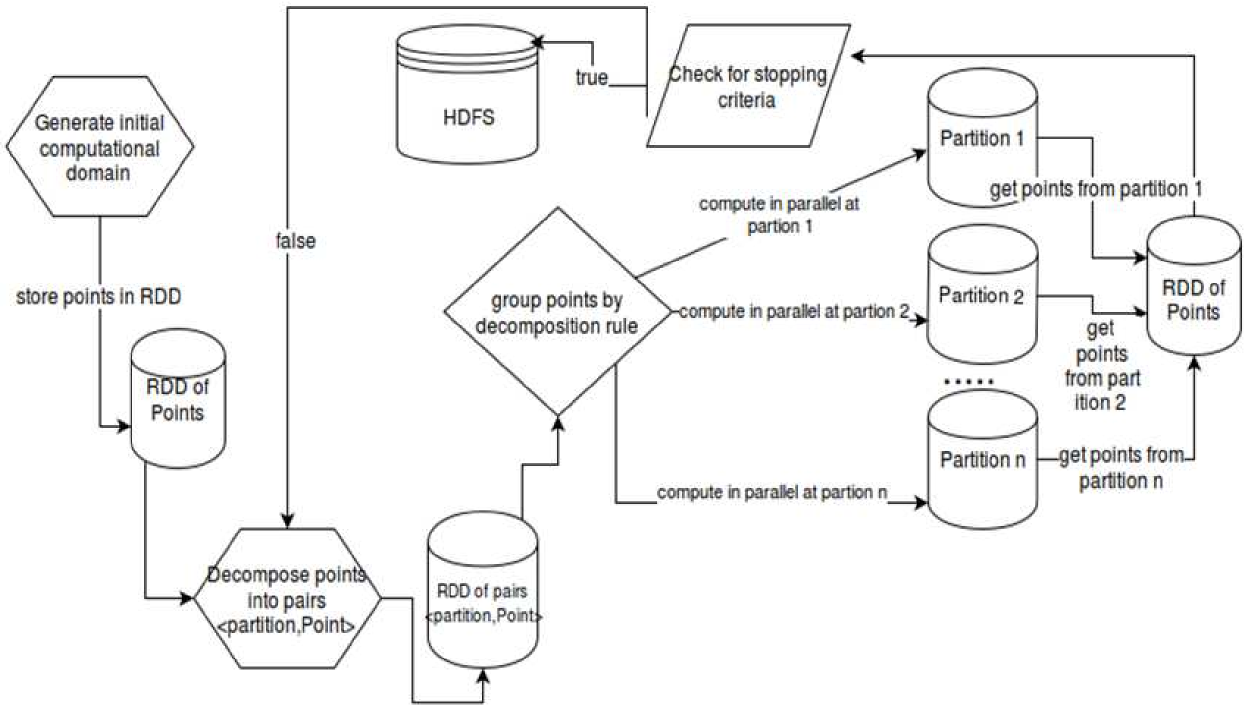
Generate initial computational domain
store points in RDD
RDD of Points
Decompose points into pairs <partition,Point>
RDD of pairs <partition,Point>
group points by decomposition rule
compute in parallel at partion 1
compute in parallel at partion 2
compute in parallel at partion n
Partition 1
Partition 2
.....
Partition n
get points from partition 1
get points from partition 2
get points from partition n
RDD of Points
Check for stopping criteria
true
false
HDFS

Fig. 3. Algorithm for computing Dirichlet problem for Poisson's equation

Experimental Results

The cluster setup consisted of the following hardware and software settings: 8 Core i-5 processor PC each equipped with 16 Gb memory cards, 2 HP Blade servers each equipped with 4 Core Intel Xeon Processors and Gigabit Ethernet switch for network connection and for all computing nodes 64-bit Ubuntu 12.10 operating system and Hadoop 2.7.2 and Spark 1.6.1 software installed.

After testing the program on cluster environment Spark implementation has been compared to Hadoop combined with MPI (Fig. 4) and Hadoop only (Fig. 5) implementations (described in (Mansurova *et al*., 2014)) in terms of running time for different sizes of computational domain and different number of cores and cluster nodes. Computational domain represents a cube with fixed and equal number of discrete points along every dimension. The results of the computation were verified for correctness compared to sequential code for the same problem with equal settings of the domain size and initial conditions. Graph below (Fig. 4) summarizes results of execution of hybrid approach that combines Hadoop and MPI for solving Dirichlet problem on single iteration of the algorithm.

Figure 5 presents the running time comparison of the Hadoop and Spark solutions for Dirichlet problem on fixed size computational domain for different number of iterations. From the figure it is clear that for higher number of iterations Spark based solution shows better running time with speed-up of up to 1.5 compared to Hadoop solution.

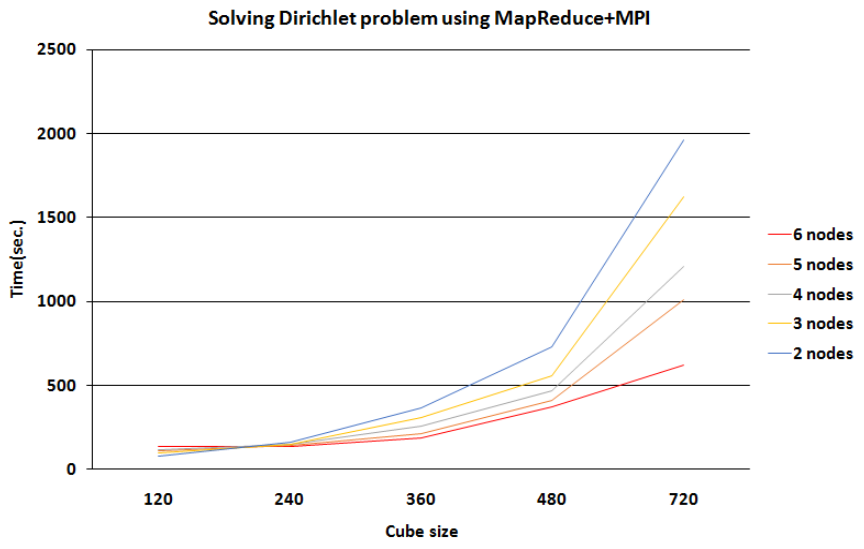

Fig. 4. Running time of Mapreduce + MPI based solution

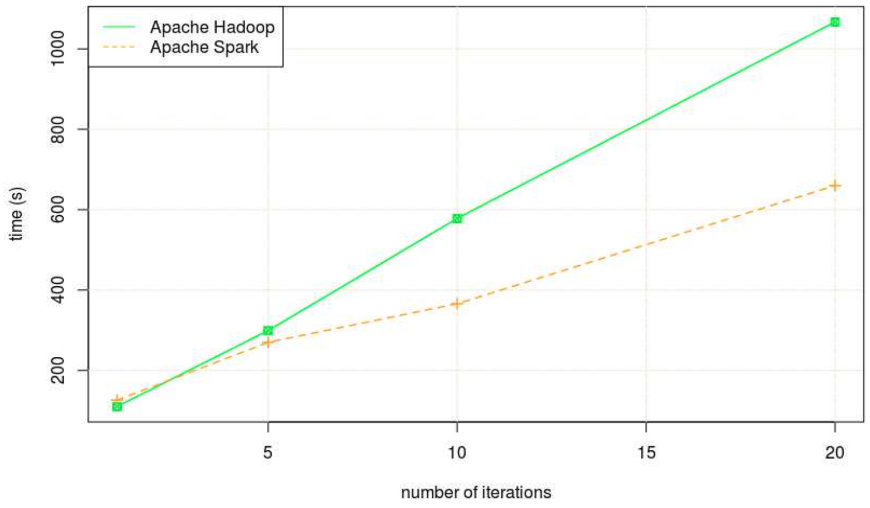

Fig. 5. Running time of Spark/Hadoop with different number of iterations on 360*360*360 points computational domain

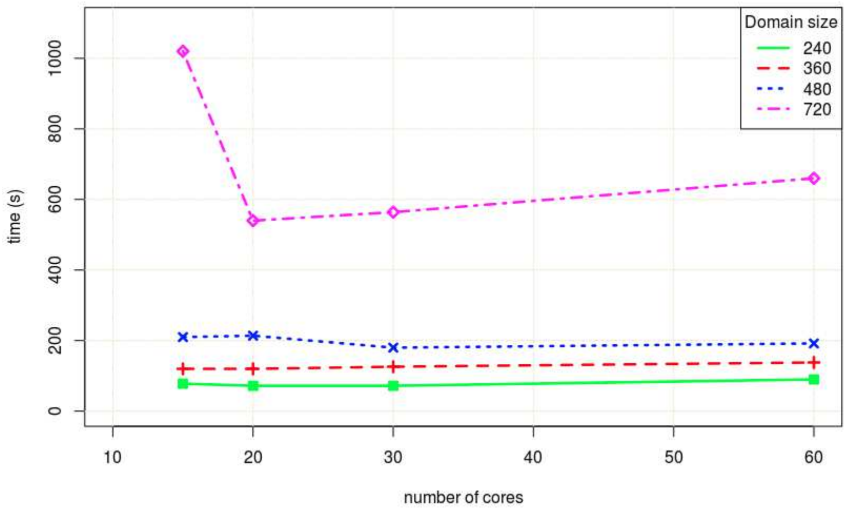

Fig. 6. Running time of Spark based solution

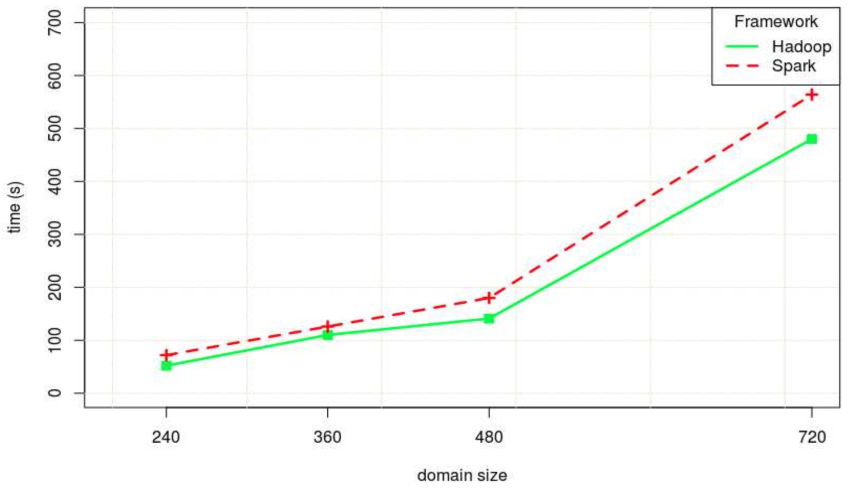

Fig. 7. Spark/Hadoop running time on 360*360*360 points computational domain

From the Fig. 6 it is easy to see that optimal running time is achieved at 30 cores setup in most cases. Increasing number of cores leads to additional overhead associated with greater network workload in exchanging boundary values from neighboring partitions in case when they located in different nodes of the cluster. Therefore results seem to reveal that Spark algorithm has some minor scalability issues in this particular use case.

Comparison of running time of Spark and Hadoop-based implementations depicted in Fig. 7 shows that Spark approach performs slightly worse in terms of running time than Hadoop-based approach on single iteration cycle. Increasing number of iterations results in performance improvement of Spark implementation compared to Hadoop-based implementation. The reason for such a performance difference is that although Spark perform more operations it perform most of the operations in memory leading to higher performance in case of large number of iterations. Hadoop, on the other hand, performs better on single iteration cycle, but in the end of the each iteration writes data back to HDFS or to local file system leading to poor performance due to accumulation of I/O latencies over the course of the computation.

Conclusion

We have designed and tested an algorithm for solving Dirichlet problem for Poisson`s equation using Apache Spark framework and compared it with Hadoop-based implementations. As a result Spark based implementation proved to be more suitable for solving Dirichlet problem due to improved performance compared to Hadoop-based implementation.

Acknowledgment

Authors would like to thank al-Farabi Kazakh National University for giving an opportunity to do current research by providing computational facilities and educational support and Ministry of Education of Kazakhstan Republic for supporting research by governmental grant.

Author's Contributions

Shomanov Aday: Author is responsible for devising and implementing the algorithm and writing the paper.

Mansurova Madina: Contributed by devising the algorithm and providing useful suggestions and corrections to the contents of the paper.

Ethics

The article contains an original and unpublished material. There is no ethical violation regarding authorship. Corresponding author confirms that all authors have read and approved the manuscript.

References

Freeman, J., 2014. Mapping brain activity at scale with cluster computing. Nat. Meth., 11: 941-950. DOI: 10.1038/nmeth.3041

Gropp, W. and E. Lusk, 2004. Fault tolerance in message passing interface programs. Int. J. High Performance Comput. Applic., 18: 363-372. DOI: 10.1177/1094342004046045

Horlacher, O., F. Lisacek and M. Müller, 2014. Mining large scale tandem mass spectrometry data for protein modifications using spectral libraries. J. Proteome Res., 15: 721-731. DOI: 10.1021/acs.jproteome.5b00877

Li, R., H. Hu, H. Li, Y. Wu and J. Yang, 2016. MapReduce parallel programming model: A state-of-the-art survey. Int. J. Parallel Programm., 44: 832-866. DOI: 10.1007/s10766-015-0395-0

Lu, X. and F. Liang, 2016. Accelerating iterative big data computing through MPI. Int. J. Comput. Sci. Technol., 30: 283-294. DOI: 10.1007/s11390-015-1522-5

Lu, X., B. Wang, L. Zha and Z. Xu, 2011. Can MPI benefit hadoop and mapreduce applications? Proceedings of 40th International Conference on Parallel Processing Workshops, Sept. 13-16, IEEE Xplore Press, pp: 371-379. DOI: 10.1109/ICPPW.2011.56

Lu, X., F. Liang, B. Wang, L. Zha and Z. Xu, 2014. DataMPI: Extending MPI to hadoop-like big data computing. Proceedings of the IEEE 28th International Parallel Distributed Processing Symposium, May 19-23, IEEE Xplore Press, pp: 829-838. DOI: 10.1109/IPDPS.2014.90

Mansurova, M., D. Ahmed-Zaki, A. Shomanov, Y. Dadykina and S. Ikhsanov *et al*., 2014. Solving dirichlet problem for poissons equation using mapreduce hadoop and MPI. Proceedings of International Conference New Trends in Information and Communication Technologies, (ICT' 14), pp: 226-234.

Reyes-Ortiz, J.L., L. Oneto and D. Anguita, 2015. Big data analytics in the cloud: Spark on hadoop Vs MPI/OpenMP on beowulf. Proc. Comput. Sci., 53: 121-130. DOI: 10.1016/j.procs.2015.07.286

Zaharia, M., M. Chowdhury, M.J. Franklin, S. Shenker and I. Stoica, 2010. Spark: Cluster computing with working sets. Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, Jun 22-25, ACM, USA, pp: 10-10.

Zaharia, M., M. Chowdhury, T. Das, A. Dave and J. Ma *et al*., 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, Apr. 25-27, ACM, USA., pp: 2-2.

Zhao, G., C. Ling and D. Sun, 2015. SparkSW: Scalable distributed computing system for large-scale biological sequence alignment. Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May 4-7, IEEE Xplore Press, pp: 845-852. DOI: 10.1109/CCGrid.2015.55