DOI: 10.1109/HiPC50609.2020.00038

Exposing data locality in HPC-based systems by using the HDFS backend

Jose Rivadeneira
Computer Science and Engineering Department
University Carlos III of Madrid
Leganes, Spain
jrivaden@pa.uc3m.es

Felix Garcia-Carballeira
Computer Science and Engineering Department
University Carlos III of Madrid
Leganes, Spain
felix.garcia@uc3m.es

Jesus Carretero
Computer Science and Engineering Department
University Carlos III of Madrid
Leganes, Spain
jesus.carretero@uc3m.es

Javier Garcia-Blas
Computer Science and Engineering Department
University Carlos III of Madrid
Leganes, Spain
fjblas@inf.uc3m.es

Abstract—**Nowadays, there are two main approaches for dealing with data-intensive applications: parallel file systems in classical High-Performance Computing (HPC) centers and Big Data like parallel file system for ensuring the data centric vision. Furthermore, there is a growing overlap between HPC and Big Data applications, given that Big Data paradigm is a growing consumer of HPC resources. HDFS is one of the most important file systems for data intensive applications while, from the parallel file systems point of view, MPI-IO is the most used interface for parallel I/O. In this paper, we propose a novel solution for taking advantage of HDFS through MPI-based parallel applications. To demonstrate its feasibility, we have included our approach in MIMIR, a MapReduce framework for MPI-based applications. We have optimized MIMIR framework by providing data locality features provided by our approach. The experimental evaluation demonstrates that our solution offers around 25% performance for *map* phase compared with the MIMIR baseline solution.**

Index Terms—**HPC, Big Data, HDFS, MPI-IO, MapReduce, Data locality.**

I. Introduction

Nowadays, there are two main approaches for dealing with data-intensive applications, parallel file systems under classical High-Performance Computing (HPC) centers and Big Data like parallel file system for ensuring the data centric vision. These file systems present different architectures, as file systems used in HPC currently employ isolated I/O nodes for accessing data, which are usually much less in number than the compute nodes. On the other hand, parallel file systems assume that each compute node has a storage device and, thus, data is spread among many (all) the compute nodes. The divergence of both kinds of file systems is mainly due to the differences between the execution framework in both environments (HPC and Big Data): while the HPC environment addresses scientific problems (problems with a high computational load), Big Data environment is specialized in processing large data sets and large amounts of files.

HPC application domains have been changing in the last years. Currently they are not only computationally expensive applications, but also generate a vast amount of intermediate data that must be optimally processed [1]. To cope with those new features, various frameworks used in Big Data world, such as Apache Hadoop or Spark [2] have been deployed on top of HPC environments. However, they have proven not to be efficient enough. For this reason, one of the main lines of research consists of unifying both environments in such a way that HPC applications can benefit from the Big Data approach and vice versa.

As discussed in article [3], the solutions used in Big Data can significantly improve the performance of the applications used in HPC, especially for data processing. One of the main paradigms used in Big Data that can help in the HPC environment corresponds to the applications of MapReduce [4]. Those application domains are mainly based on the usage of data locality to obtain a significant improvement of performance [5]. One of the most used frameworks for running MapReduce applications corresponds to Hadoop and its associated HDFS filesystem [6].

The usage of Big Data storage would allow to overcome one of the main problems existing in the HPC environment: the bottleneck caused by access to data. As some authors have proposed, making use of data locality information, as occurs in Big Data, would be instrumental to provide in-situ data processing, avoiding sending the data to be processed to the compute nodes [7]. Most of the applications designed for HPC make use of MPI and the MPI-IO input/output interface [8], which does not provide functions to gather a node list where data are stored. Moreover, due to the HPC systems architecture, most existing parallel file systems are incompatible for

This work was partially supported by the project “CABAHLA-CM: Convergencia Big data-Hpc: de los sensores a las Aplicaciones” S2018/TCS-4423 from Madrid Regional Government and the European Union’s Horizon 2020 research, New Data Intensive Computing Methods for High-End and Edge Computing Platforms (DECIDE). Ref. PID2019-107858GB-I00 and innovation program under grant agreement No 801091, project “ÀSPIDE: Exascale programming models for extreme data processing”.

the location of data in storage, thus preventing to make in-situ data processing as a general rule.

For those reasons, some authors have proposed to change the architecture of HPC environments to include local storage [9], [10]. This is possible nowadays thanks to Solid State Disks (SSD) devices and, in a near future, the non-volatile memory (NVM) and the increase of their storage capacity. In fact, some supercomputers within the top 500 [11] already offer a large capacity of local storage, and therefore they can mount a distributed file system on them. Moreover, the locality of the data can also be used in the event that a hybrid system consisting of parallel file systems and distributed file systems is available, as intermediate readings and writes can write to the distributed file system using locality. However, most parallel applications are programmed using the MPI standard, making I/O operations using the MPI-IO interface or directly through the POSIX interface, and any of those interfaces provide facilities currently to exploit data locality.

In an attempt to solve the aforementioned problems and challenges, in this work we present an extension of the MPI interface that enables to know and exploit the location of the data in the storage subsystem. We also present an extension of the MPI-IO interface to support HDFS, a file system widely used in Big Data applications. To conclude, we have evaluated our solution comparing it with MIMIR, a MapReduce based framework that support MapReduce on top of the MPI runtime.

The structure of the paper is as follows. Section II presents work related to the research in this paper. Section III introduces the design and implementation of our contributions. The evaluation of these contributions is shown in Section V. Finally, Section VI concludes this paper by discussing about the contributions and enumerating of future work.

II. Related Work

MapReduce is a very popular paradigm in Big Data [12], [13] as it is designed to solve Single Program Multiple Data (SPMD) problems that are very common domains like data analytics. However, MapReduce was designed considering and underlying distributed infrastructure, including local storage and a distributed file system. An example of this is HDFS [14], which provides multiple features, including data replication. Following this popularity, some works in the HPC domain have provided MapReduce based implementations suitable for HPC supercomputers [15], [16]. MR-MPI [17] was one of the first initiatives to provide a MapReduce library on top of MPI. However, some other authors have claimed that this solution suffers from scalability limitations. Most recently, MIMIR [18] has been proposed as a tool to provide memory-efficient and scalable MapReduce for large supercomputing systems. A common problem of those solutions is that data location is not considered, which limits the overall I/O performance. In contrast, our solution targets to improve the I/O performance by take advantage of the data locality features provided by HDFS.

Various authors have proposed different solutions for improving the performance of MapReduce applications in the HPC environment [19], [20]. The work proposed by Unat et al. [7] discussed about several ways in which we can exploit the locality of data in HPC environments by both compiler and runtime. From those, making use of data locality to avoid the bottleneck at the entrance and output following the Big Data model has become a feasible solution [21].

In the literature, we found authors that have proposed solutions to know the data localization in HPC environments, making use of multiple implementations based on MPI. As an example, Lu et al. [22] provided an MPI adaptation based on the Hadoop environment. In [23], the authors presented an implementation that works at different levels of the system, including both memory and storage levels. In other articles, authors implement a locality layer between MPI and HDFS [24], so that they can locally access the data. In our proposed work, we will focus on the way to exploit the locality at system level, either by means of memory (in-memory storage [25]) or by means of the local storage available at the computing nodes.

As far as we know, all existing solutions require custom implementations to make use of data locality, which forces the users to adapt their applications to deal with the new framework. Therefore, in this work we propose the extension of the MPI interface in such a way that it allows to exploit data locality on those file systems supporting it. In this way the door can be opened to be able to use the locality at the system level to all the file systems that are implemented in MPI, whether they are distributed file systems, parallel file systems, in-memory systems or burst-buffers.

III. Data locality extension for MPI-IO

One of the main objectives of this paper is to improve the way in which MPI accesses data, exploiting the data locality mechanisms provided by the back-end file system. To demonstrate this functionality, this work present three main contributions. First, extending MPI interface for data locality; Second creating an MPI-IO connector to HDFS into ADIO; Finally, we have adapted MIMIR to use the data locality facilities proposed. Our first contribution is to expand the MPI-IO interface with two new functions that provide information on the location of the data stored. The functions that we propose are shown below (Functions 2 and 1).

Function 1 Get file data block size

```
    Get file data block size

    MPIX_File_get_blocksize(MPI_File , int * )
Require: MPI_File : File descriptor.
Ensure: int *: File block size in bytes. If this feature is not
    available a negative value is returned.
```

Function 2 Get file data location

Get file data location

MPIX_File_get_locality(MPI_File , MPI_Offset , MPI_Offset , char** , int *)**

Require: MPI_File : File descriptor

Require: MPI_Offset : Initial offset of the buffer to be located

Require: MPI_Offset : Final offset of the buffer to be located

Ensure: char**** : List of nodes where blocks are located, if locality is available in the file system. NULL if locality is not available in the file system

Ensure: int *replication: 0 if locality is not available in the file system, else a value greater than 0 showing the replication level in the file system.

For the first function, the second argument returns the block size in bytes of the file indicated by the first argument.

The fourth and fifth arguments of the second function return two values. Firstly, the fourth argument returns the identity of the node(s) where the file block(s) storing the offset interval defined in arguments 2 and 3 is stored for the file indicated by the first argument, the identity of the node returned is the same as the value returned by the function MPI_Get_proccesor_name. Secondly, the number of replicas used to store the blocks of the file is returned in the fifth argument. Using this function, any application using MPI could know the node where data is located and take placement decisions to make in-situ data processing.

Listing 1 shows an example of usage of the functions proposed to get the locations of the blocks of a file. As may be seen, in spite of its power, the usage of the functions inside an MPI program is very easy and not disruptive. Of course, the functions depend on the file system capacity to support locality and replication features.

Listing 1: Demo of the usage of the locality functions

```c

#include <stdio.h>
#include "mpi.h"

int main(int argc, char * argv[]){
  MPI_File fd;
  MPI_Offset filesize;
  int blocksize;
  int replication;
  char *** nodes = NULL;

  MPI_Init(&argc,&argv);

  // Enable locality support
  MPI_Info info;
  MPI_Info_create(&info);
  MPI_Info_set(info,"hdfs_set_locality","1");
  // Open file for reading
  MPI_File_open(MPI_COMM_WORLD,"hdfs://filename",
       MPI_MODE_RDONLY,info,&fd);

  //Get file size
  MPI_File_get_size(fd,&filesize);
  //Get the size of the file block
  MPIX_File_get_blocksize(fd,&blocksize);
  if(blocksize < 0){
    // Feature not supported by FS
    printf("Get block size failed \n");
    MPI_Abort(MPI_COMM_WORLD,-1);
  }

  //Get the id of the nodes where each block is stored
  MPIX_File_get_locality(fd,0,filesize,&nodes,&
       replication);
  if(nodes == NULL){
    // Locality not supported by FS
    printf("Get locality failed \n");
    MPI_Abort(MPI_COMM_WORLD,-2);
  }

  // Number of blocks in the file.
  int blocksnumber = filesize / blocksize;
  //Demo loop to print the nodes where
  // each block is stored
  for(int i = 0; i < blocksnumber;i++){
    for(int j = 0; j < replication ;j++){
      printf("BLOCK::%d stored in the node :: %s\n",i,
           nodes[i][j]);
    }
  }
  //Close file
  MPI_File_close(&fd);
  MPI_Finalize();
  return 0;
}
```

A. Supporting HDFS in MPI-IO

After including the proposed locality mechanisms inside MPI-IO, we created a new file system connector for HDFS. We have employed MPICH [26], as this version of MPI provides access to several file systems through ROMIO [27] and the abstract device interface for parallel I/O, called ADIO [28]. Figure 1 shows the structure of the MPI-IO stack after the inclusion of the HDFS connector presented in this paper.

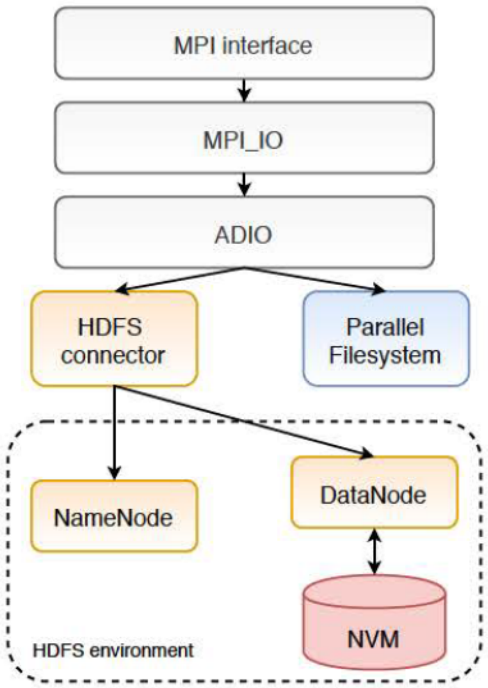

Fig. 1: MPI-IO stack including HDFS connector.

At file creation in HDFS, the interface enables the definition of parameters such as block size, buffer size, and the number of replicas for each block. It is also possible to create files only for writing or it will be later available for reading. In order to implement all these features, we have made use of the `MPI_Info` abstraction. The following variables have been defined in `MPI_Info` to permit users to customize the way of creating or writing files in HDFS:

- `write_mode`: allows to specify when data written will be available for the user. By default, `WRITE_ON_CLOSE` policy is employed, but also, users can define `HFLUSH` to enforce immediate writing to storage (write-thought).
- `hdfs_buffersize`: allows to define the size in bytes for the internal HDFS buffer used to temporally store file system block in main memory of the process.
- `hdfs_replication`: permits to define the replication level (number of replicas) for the file created.
- `hdfs_blocksize`: defines the size in bytes of the file block.

MPI-IO has a large number of functions that enable access to different file systems. Because HDFS implements single-writer multiple-readers model, it has not been possible to implement all the MPI functionalities. We list the recognized limitations:

- A file can be only be open in write or read mode, never read/write.
- Concurrent writes to a file is not allowed.
- Non-blocking read/write functions have not been implemented in the current prototype.

IV. MIMIR INTEGRATION OF HDFS

Currently, there are multiple frameworks that permit the execution of Big Data applications under MPI environments. To check how the effect of data locality in one of those frameworks, we have adapted the current implementation of MIMIR to use HDFS and our locality functions.

The current implementation of MIMIR performs a block distribution algorithm without taking into account the file system below, as it assumes that the underlying file system will read the blocks wherever they are and will bring them to the compute nodes. Thus, each worker reads a consecutive chunk of the file, without considering any locality. However, having a HDFS distributed file system, part of the blocks to read could be stored in the same compute node.

In this work, we have modified MIMIR's block distribution algorithm in such a way that each worker only processes those file blocks which are stored locally, as shown in Figure 2. In this way, we reduce the amount of data sent over the network, sending only the necessary data for the shuffle phase. For this purpose, we have made use of the MPI functions designed and previously implemented, as well as the HDFS connector.

One of the main features of distributed file systems is the block replication, which means that the same block is stored on multiple storage nodes simultaneously. Our algorithm takes advantage of this feature in order to enhance the distribution of the workload among the nodes in case the file system does not distribute the blocks evenly (i.e., HDFS).

Another important feature of our approach is how the block assignation between the computation nodes is done. During this work, in order to have a homogeneous distribution of the workload among the different processes that are running on a given node, the block allocation is done per node, instead of per process. This implies that each assigned block to a certain machine will be divided among the different processes that are running on that node as shown in the Figure 2.

Listing 2: MIMIR wordcount example.

```
1   //Map function
2   void map(Readable<char *, void> *input, Writable<char *,
        uint64_t> *output, void *ptr)
3   {
4       char *line = NULL;
5       while (input->read(&line, NULL) == true)
6       {
7           char *saveptr = NULL;
8           char *word = strtok_r(line, " ", &saveptr);
9           while (word != NULL)
10          {
11              if (strlen(word) < 1024) {
12                  uint64_t one = 1;
13                  output->write(&word, &one);
14                  nwords += 1;
15              }
16              word = strtok_r(NULL, " ", &saveptr);
17          }
18      }
19  }
20
21
22
23  //Reduce function
24  void reduce(Readable<char *, uint64_t> *input, Writable<
        char *, uint64_t> *output, void *ptr);
25  {
26      char *key = NULL;
27      uint64_t val = 0;
28      uint64_t count = 0;
29      while (input->read(&key, &val) == true)
30      {
31          count += val;
32      }
33      output->write(&key, &count);
34  }
```

Listing 2 shows a fragment of the classical wordcount use case implemented on top of MIMIR [1]. As shown in Line 5, the read function is internally implemented by using the MPI-IO interface. In this case, the *read* call individually extracts each line of the file in an iterative manner. In the past, this file access was done using the parallel system provided by the infrastructure (i.e., Lustre, GPFS, OrangeFS). This approach has two drawbacks. First, it is difficult to tune the accessed block size for adapting the nature of the input data. Second, the source code is executed in the compute node in where the data are located.

To cope with those aforementioned counterparts, we have extended MIMIR to support data locality and to fine-tune the HDFS locality-aware file system. Algorithm 1 details the steps need to provide a data-locality scheme inside MIMIR framework.

Another extension corresponds to the sharing of the end of the block between different processes. In a similar way to MIMIR, in our implementation we have decided to tokenize

[1]This source code is taken from the MIMIR Git repository at https://github.com/TauferLab/Mimir/blob/master/examples/wordcount.cpp

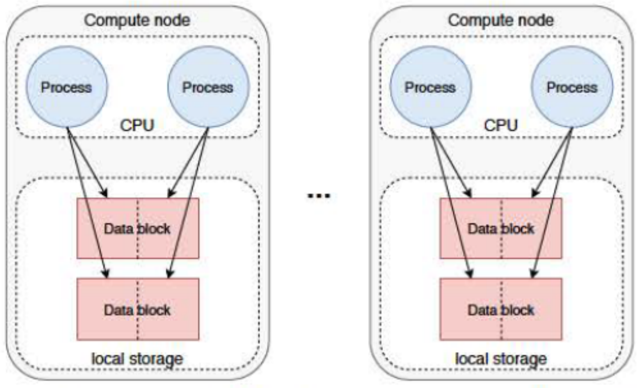

Fig. 2: Distribution of blocks between processes with locality.

Algorithm 1 Algorithm to include data locality in MIMIR

```
Require: File in HDFS to be processed.
Ensure: List of blocks to be processed by each process.
 1: for all Running process do
 2:     Send all to all the name of the node where each process is running.
 3: end for
 4: Get the file size.
 5: Get the block size of the file.
 6: Calculate the number of blocks in the file.
 7: for all Blocks file do
 8:     Get the machines in which the block is stored.
 9:     for all Machines do
10:         The machine with the fewest allocated blocks is selected.
11:         Get the number of processed which are running in this machine.
12:         if There is only one process executing in that machine then
13:             The block is assigned to that process.
14:         else
15:             The block is divided between the different processes running on that machine.
16:         end if
17:     end for
18: end for
19: return List of blocks processes by each process.
```

the file into lines. To solve the problem that a token cannot be divided into two blocks, we propose a mechanism divided into the following steps:

1) Each process reads the portion of the block allocated to it and a fragment of the consecutive portion (even if it is another block).
2) Each process processes the first token found in the consecutive portion.
3) If it is not the first fragment of the first block, each process discards the first token of its portion, since it has been processed by the previous fragment.

V. Evaluation

The experimental evaluation has been carried out in a cluster of 8 nodes with the following configuration. Each node has an Intel(R) Xeon(R) CPU E5-2603 v4 processor with 126GB DDR4 of RAM memory. Additionally, each node has two hard disks, one with a capacity of 512 GB for storage of the operating system and applications and the other with a capacity of 2TB for data storage. All nodes are connected through a 1 Gbps network interface. The compiler used is GCC 8.0. After that, the source code has been compiled using both `-O3` and `-DNDEBUG` flags. HDFS storage system is configured to use 64 MB blocks and all nodes act as *DataNode*. The result shown in this paper are obtained by calculating the averaged value of five consecutive executions. The MPI implementation employed in this paper is MPICH 3.2.

In order to evaluate the performance of our proposed interface, we have carried out two type of experiments. First, we have evaluated the overhead of our proposed ROMIO interface in comparison with the native C-based HDFS interface called libhdfs[2]. Second, we evaluated how the implementation of our locality algorithm affected the performance of the overall applications. To conduct these experiments, two applications have been developed on top of the MIMIR framework: word-count (wc) and a protein matching (pm) [29] applications.

A. HDFS interface for MPI-IO

We have designed different testbeds to evaluate the overhead built into our HDFS interface implementation. Figures 3, 4 and 5 plot the overall throughput by reading a single file of 1, 8 and 32 GiB. *libhdfs* corresponds to native HDFS functions, implicit offset corresponds to MPI_File_read or MPI_File_write, explicit offset corresponds to MPI_File_read_at and finally shared pointer corresponds to MPI_File_read_shared. In addition, we also modify the size of the read and write request (1 and 128 MB), with the aim of modifying the number of requests made by each process performs.

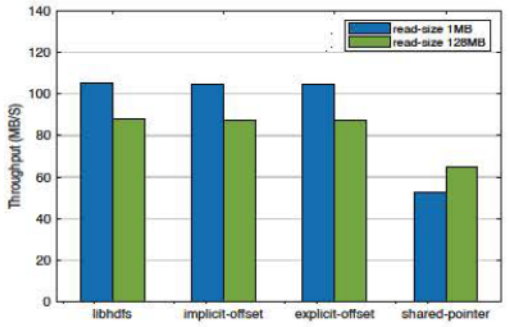

Fig. 3: Throughput obtained reading an 8GB file.

[2]A JNI based C API for HDFS https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/LibHdfs.html

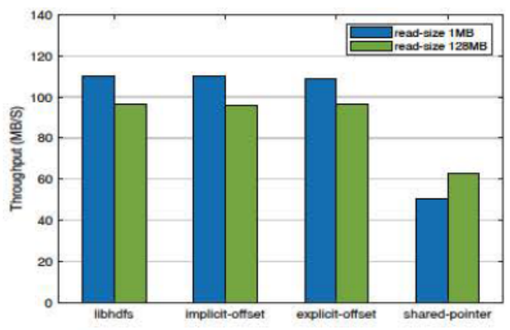

Fig. 4: Throughput obtained reading a 32GB file.

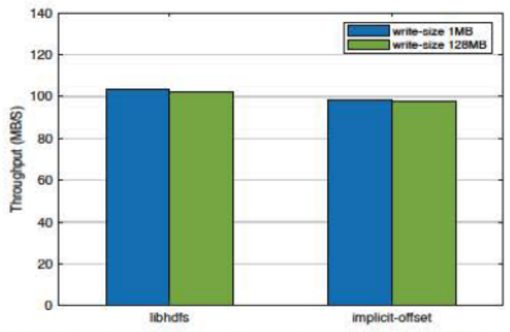

Fig. 5: Throughput obtained writing a 1GB file.

As can be seen in Figures 3 and 4, the functions MPI_File_read and MPI_File_read_at do not incorporate any overhead in the read time with respect to the native HDFS library to perform the same operation. We also note that the function MPI_File_read_shared does not scale with respect to the number of read requests that are made. This is mainly due to the way the processes share the file pointer in our implementation.

We have also evaluated the performance of the write function MPI_File_write with respect to the native HDFS write function. Due to HDFS disallows simultaneous writing of a file by multiple processes, this evaluation is limited to write a 1 GB file by a single process. Figure 5 depicts that no overhead has been added in the implementation of the write function with respect to the native functions of HDFS.

B. Using MIMIR locality with HDFS

Currently, there are a large number of applications that take advantage of the MapReduce paradigm [30]. To study the effects of data locality on the runtime of the applications, we have designed two Big Data applications using the MIMIR framework: (1) wordcount (wc), which counts the number of occurrences of a word in a file; and (2) a protein matching application (pm), which searches a certain protein in a large dataset. Both applications have been selected to compare the data locality effect on applications that generate a large number of <key, value>pairs and on those whose number of generated pairs is lower.

We carry out two different experiments processing with two different datasets (8GB and 32 GB) using from 8 to 64 processes. Moreover, because HDFS does not distribute blocks in storage nodes uniformly, we have configured two different experiments per application varying the blocks replication factor from 1 to 3. In this work, we present the results obtained in the Map phase.

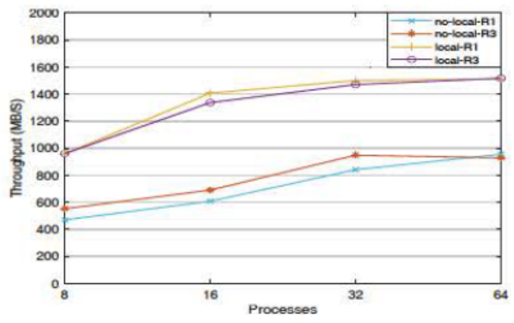

Fig. 6: Throughput reading an 8GB file in wc application

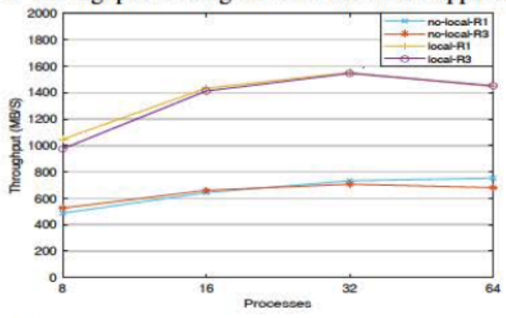

Fig. 7: Throughput reading a 32GB file in wc application

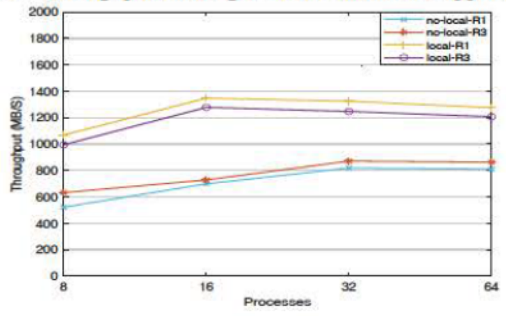

Fig. 8: Throughput reading an 8GB file in pm application

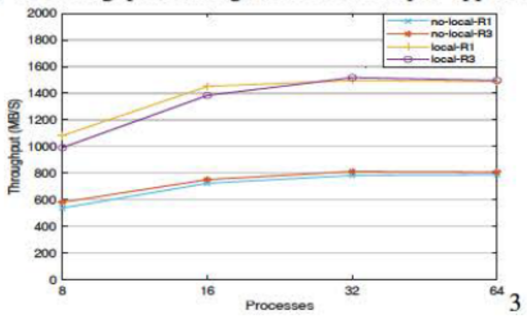

Fig. 9: Throughput reading a 32GB file in pm application

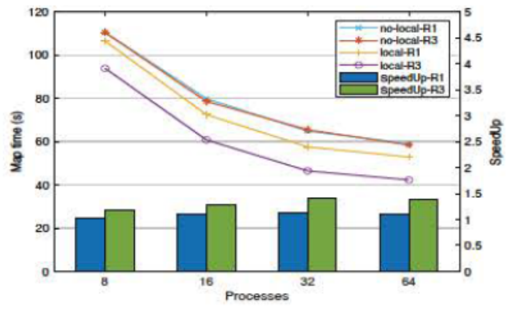
Fig. 10: Map execution time comparison for wc with 8GB

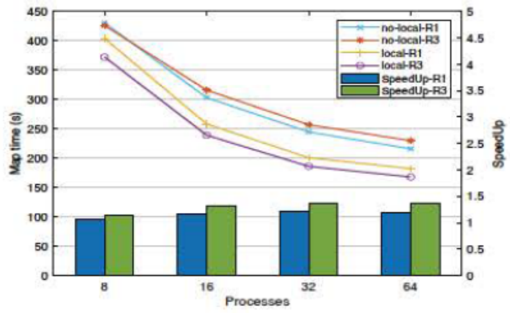
Fig. 11: Map execution time comparison for wc with 32GB

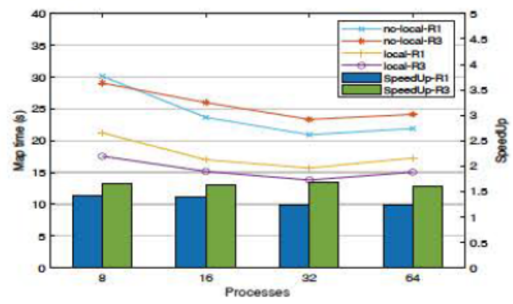
Fig. 12: Map execution time comparison for pm with 8GB

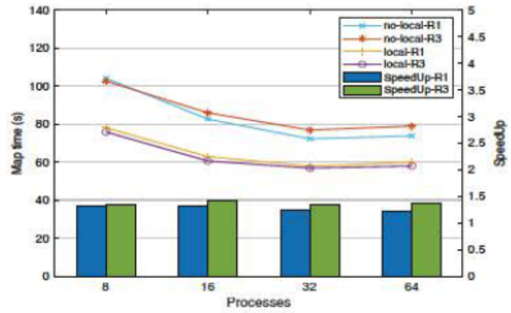
Fig. 13: Map execution time comparison for pm with 32GB

We measured the effect of data locality on read operations of the Map process for both applications wc and pm. Figures 6, 7, 8, and 9 show the results obtained. As may be seen in case of wc application, the throughput to access data increases by 50% when our data locality solution is applied. Figure 9 plots how the data access performance is also better for the pm application. We observe that in case of wc application, our approach gets better performance improvement than pm. This is mainly motivated by the fact that the employed network setup shares the bandwidth for both data and computation. We also highlight that the number of <key, value> pairs is much lower is case of pm, which produces fewer intermediate data, reducing the stress in the network capacity and reducing the influence of that bottleneck. Finally, we conclude that the inclusion of data replication does not reduce the average time each process spent on each read operation.

Figures 10 and 11 show the Map phase execution time for the wc application with and without applying data locality for 8GB and 32GB files (lines) and the speedup obtained with our solution (bars). As may be seen, the total time required by the Map phase is reduced in all cases. The replication factor plays a significant role in the total execution results, as shown in the figure, because with a replication factor of 3 we achieve even more speedup than with replication factor of 1, This is due to the better distribution of the workload achieved by the proposed solution with a higher replication. The improvement in the execution time of the Map phase is not only due to the reduction in the access time to the data. In the first version of MIMIR, the blocks of the files had to be sent over the network, as well as all the <key, value> pairs that each of the processes generated. Thus, including locality reduces the amount of data that must travel over the network, the bottleneck that it must support is significantly reduced.

Figures 12 and 13 show the results for the pm application. It presents similar results as the wordcount application. This application requires less computation time compared to the wordcount application, so reading time occupies a higher percentage of the total execution time of the Map phase. For this reason, it is observed that a better speedup is achieved under equal conditions.

VI. Conclusions and Future Work

In this paper, we have proposed two new functions for MPI-IO that are able to extract the data locality of a file, assuming that the underlying file system provides those details. To exploit locality functions we have created a new MPI-IO connector for HDFS that has been deployed in the MPICH distribution.

To demonstrate its feasibility, we have evaluated our proposal including it into MIMIR, a framework for MapReduce for MPI applications, and we have shown that MIMIR framework can be optimized including data locality support, information that is obtained through our MPI-IO implementation. We have also demonstrated that the proposed functionality enhances MIMIR performance for two typical Big Data applications, such as wordcount and protein matching, as the read data phase is greatly reduced by using our functions.

In our opinion, this locality functions would be beneficial or most applications running in large scale system from now on as in-memory storage systems and large size SSD will provide local storage per node, but also for some file system providing some locality info such us HDFS, Luster, etc. In general, knowing where data are, will allow applications to include strategies to reduce the I/O time for Big Data applications running in HPC systems and for workflows crating data dependencies.

Work is going on to make a proposal to include the functionality in the MPI standard, to create more connectors in ADIO to provide data locality information, and to extend and enhance the HDFS interface to support as many MPI-IO functionalities as possible (for example, collective writing on a file or file pointer sharing).

REFERENCES

[1] D. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, pp. 56–68, 07 2015.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[3] S. Caíno-Lores, J. Carretero, B. Nicolae, O. Yildiz, and T. Peterka, "Toward high-performance computing and big data analytics convergence: The case of spark-diy," *IEEE Access*, vol. 7, pp. 156 929–156 955, 2019.

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[5] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. USA: IEEE, May 2012, pp. 419–426.

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The HADOOP distributed file system," in *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies*. Sunnyvale, California USA: IEEE, 2010, pp. 1–10.

[7] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericás, "Trends in data locality abstractions for hpc systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 3007–3020, Oct 2017.

[8] MPI Forum, "Mpi: A message-passing interface standard," University of Tennessee, Knoxville, TN, USA, Tech. Rep., 1994.

[9] F. Isaila, J. Garcia Blas, J. Carretero, R. Ross, and D. Kimpe, "Making the case for reforming the i/o software stack of extreme-scale systems," *Advances in Engineering Software*, vol. 111, pp. 6–31, 07 2017.

[10] I. Raicu, I. Foster, and P. Beckman, "Making a case for distributed file systems at exascale," *LSAP'11 - Proceedings of the 3rd International Workshop on Large-Scale System and Application Performance*, pp. 11–18, 01 2011.

[11] E. Strohmaier *et al.* (2019) 54th edition of the top500 list. https://www.top500.org/lists/2017/06/. Top500. (Acceso: 22-05-2019).

[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] ——, "Mapreduce: a flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[14] F. Azzedin, "Towards a scalable hdfs architecture," in *2013 international conference on collaboration technologies and systems (CTS)*. IEEE, 2013, pp. 155–161.

[15] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan, "Mariane: Using mapreduce in hpc environments," *Future Generation Computer Systems*, vol. 36, pp. 379–388, 2014.

[16] Y. Guo, W. Bland, P. Balaji, and X. Zhou, "Fault tolerant mapreduce-mpi for hpc clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[17] C. Engelmann and S. Böhm, "Redundant execution of hpc applications with mr-mpi," in *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, 2011, pp. 15–17.

[18] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "Mimir: Memory-efficient and scalable mapreduce for large supercomputing systems," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 1098–1108.

[19] M. Wasi-ur Rahman, N. S. Islam, X. Lu, and D. K. D. Panda, "A comprehensive study of mapreduce over lustre for intermediate data placement and shuffle strategies on hpc clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 633–646, 2016.

[20] J. Veiga, R. R. Expósito, G. L. Taboada, and J. Touriño, "Analysis and evaluation of mapreduce solutions on an hpc cluster," *Computers & Electrical Engineering*, vol. 50, pp. 200–216, 2016.

[21] N. S. Islam, M. Wasi-ur-Rahman, X. Lu, and D. K. D. K. Panda, "Efficient data access strategies for hadoop and spark on hpc cluster with heterogeneous storage," in *2016 IEEE International Conference on Big Data (Big Data)*. USA: IEEE, 12 2016, pp. 223–232.

[22] 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*. IEEE, 2014, pp. 829–838.

[23] T.-C. Dao and S. Chiba, "SEMem: Deployment of MPI-Based In-Memory Storage for Hadoop on Supercomputers," in *2018 IEEE 20th International Conference on High Performance Computing and Communications*. USA: IEEE, 08 2017, pp. 442–454.

[24] J. Yin, A. Foran, and J. Wang, "DL-MPI: Enabling data locality computation for MPI-based data-intensive applications," in *Proceedings of the 2013 IEEE International Conference on Big Data, Big Data 2013*. USA: IEEE, 10 2013, pp. 506–511.

[25] F. R. Duro, F. Marozzo, J. G. Blas, D. Talia, and P. Trunfio, "Exploiting in-memory storage for improving workflow executions in cloud platforms," *The Journal of Supercomputing*, vol. 72, no. 11, pp. 4069–4088, 2016.

[26] W. Gropp, "Mpich2: A new start for mpi implementations," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2002, pp. 7–7.

[27] R. Thakur, R. Ross, E. Lusk, W. Gropp, and R. Latham, *Users guide for romio: A hihg-performance portable mpi-io implementation*, Argonee National Labs, Abril 2010.

[28] R. Thakur, W. Gropp, and E. Lusk, "An abstract-device interface for implementing portable parallel-i/o interfaces," in *gProceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computing, 1996*. USA: IEEE, 11 1996, pp. 180–187.

[29] D. Mrozek, *Scalable big data analitycs for protein bioinformatics*. Springer, 2018, vol. 28.

[30] C. Luo, W. Gao, Z. Jia, R. Han, J. Li, X. Lin, L.Wang, Y. Zhu, and J. Zhau, "Handbook of bigdatabench (version 3.1)-a big data benchmark suite," Institute of computing technology and Chinese academy of science, No.6 Kexueyuan South Road Zhongguancun,Haidian District Beijing,China, Tech. Rep.