Collective Computing for Scientific Big Data Analysis

Jialin Liu
Department of Computer Science
Texas Tech University
Lubbock, TX, 79409
Email: jaln.liu@ttu.edu

Yong Chen
Department of Computer Science
Texas Tech University
Lubbock, TX, 79409
Email: yong.chen@ttu.edu

Surendra Byna
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA, 94720
Email: sbyna@lbl.gov

***Abstract*—Big science discovery requires an efficient computing framework in the high performance computing architecture. Traditional scientific data analysis relies on Message Passing Interface (MPI) and MPI-IO to achieve fast computing and low I/O bottleneck. Among them, two-phase collective I/O is commonly used to reduce data movement by optimizing the non-contiguous I/O pattern. However, the inherent constraint of collective I/O prevents it from having a flexible combination with computing and lacks an efficient non-blocking I/O-Computing framework in current HPC. In this work, we propose *Collective Computing*, a framework that breaks the constraint of the two-phase collective I/O and provides an efficient non-blocking computing paradigm with runtime support. The fundamental idea is to move the analysis stage in advance and insert the computation into the two-phase I/O, such that the data in the first I/O phase can be computed in place and the second shuffle phase is minimized with a reduce operation. We motivate this idea by profiling the I/O and CPU usage. With both theoretical analysis and evaluation on real application and benchmarks, we show that the collective computing can achieve 2.5X speedup and is promising in big scientific data analysis.**

***Keywords*-collective computing; big data; map reduce**

I. Introduction

In current scientific applications, big data solution is recognized as the key to big science and big discovery. Though the software and hardware have been largely improved to keep up with the big data problem, it still faces challenges as the data volume increases. The source of scientific data includes instruments, simulations, and applications. Scientists face more challenges in discovering knowledge with increasing data size. For example, the Large Synoptic Survey Telescope (LSST), which is scheduled to go live in 2020, will feature a 3.2-gigapixel camera capturing ultra-high-resolution images of the sky every 15 seconds, every night, for at least 10 years [1]. Table I shows the data requirements of representative scientific applications run at Argonne Leadership Computing Facility (ALCF) through the DOE's INCITE program several years ago[19]. The data volume processed online by many applications has exceeded TBs or even tens of TBs; the off-line data is near PBs of scale.

I/O bottleneck continues to dominate the data analysis performance in most scientific applications. The major reason is that scientific data processing generates large amounts of non-contiguous I/O requests and often involves huge data movements from storage to compute nodes [7, 12, 23]. The traditional two-phase collective I/O has been commonly used to optimize the non-contiguous I/O in scientific data analysis [5, 22]. It has also been integrated in most popular high-level I/O libraries, e.g., HDF5 [2, 6, 14, 25] and PnetCDF [15, 21]. This I/O strategy performs effectively in addressing the non-contiguous data access in scientific applications. However, the data movement is still challenging. The issues come from the inherent two-phase design. The traditional two-phase collective I/O is only motivated from an I/O perspective. Such design abstracts the I/O layer that delivers high bandwidth I/O for applications; however it also prevents it from efficient combination with the analysis stage. The analysis stage, which happens after the I/O stage, must wait for the completion of I/O in a blocking way. The non-blocking collective I/O [4, 24, 27] touches this issue, but in a coarse granularity and does not address it well. There lacks a flexible computation paradigm with collective I/O.

Existing related work also includes node reordering [3], dynamic file domain partitioning [13], and inter-server coordination [28], etc. These works optimized the collective I/O by considering the network topology, data locality and concurrency. However, few of them tackle the two-phase constraint and enhance it beyond I/O perspective. In this

Table I: Data Requirements of Representative INCITE Applications at ALCF [19]

Project	On-Line Data	Off-Line Data
FLASH: Buoyancy-Driven Turbulent Nuclear Burning	75TB	300TB
Reactor Core Hydrodynamics	2TB	5TB
Computational Nuclear Structure	4TB	40TB
Computational Protein Structure	1TB	2TB
Performance Evaluation and Analysis	1TB	1TB
Climate Science	10TB	345TB
Parkinson's Disease	2.5TB	50TB
Plasma Microturbulence	2TB	10TB
Lattice QCD	1TB	44TB
Thermal Striping in Sodium Cooled Reactors	4TB	8TB

paper, we provide a novel computing paradigm for scientific applications based on the two-phase collective I/O. This computing paradigm integrates the collective I/O with computation into a paradigm, i.e., 'collective computing'. The fundamental idea is to split the collective I/O's 'two-phase', and proactively conduct the computation onto the collected segments such that the analysis stage is moved in advance of the second phase, and the second phase is optimized with reduced data size. The intermediate computation results are distributed to other nodes during the second phase. After the two-phase I/O, we can have partial results ready for the future analysis stage. The analysis stage becomes a lightweight results reduction process.

It is the first time to integrate the analysis stage within the collective I/O, and the idea of fusing computation and I/O provides a highly efficient mapreduce-like paradigm while maintaining MPI's advantages. The contributions of this work include:

- We break the two-phase I/O constraint and form a flexible collective computing paradigm.
- We propose 'object I/O' to integrate the analysis task within the collective I/O.
- We design 'logical map' to recognize the byte sequence among I/O phases and iterations.
- We analyze the collective computing experimentally using both benchmark and real application.

In the rest of the paper, we first introduce the collective I/O and motivate our idea using an initial profiling in Section II-A. We then briefly discuss the potential of applying a mapreduce paradigm and compare it with collective I/O in the remaining subsections of Section II. We discuss the design of collective computing in Section III. We evaluate the framework in Section IV and discuss the related work in Section V. We conclude the work in Section VI.

II. Motivation

A. Two-Phase Collective I/O

MPI is the dominant parallel programming model on all large-scale parallel machines, such as Cray XT5/XK6/XK7, IBM Blue Gene/P, and IBM Blue Gene/Q supercomputers. MPI-IO is a subset of the MPI-2/MPI-3 specification [8]. It defines an I/O access interface for parallel I/O. ROMIO is a popular MPI-IO implementation [22]. It provides an abstract-device interface called ADIO for implementing the portable parallel I/O API. It performs various optimizations, including collective I/O and data sieving, for common access patterns of parallel applications. Collective I/O is one of the most important I/O access optimizations. In collective I/O, multiple processes cooperate with each other to carry out large aggregated I/O requests, instead of performing many non-contiguous and small I/Os independently. A widely-used implementation of collective I/O is the two-phase I/O protocol [22]. This strategy serves the I/O requests using an I/O phase and a data exchange/shuffle phase. In the case of two phase collective read, the first phase consists of a certain number of processes that are assigned as aggregators to access large contiguous data. In the second phase, those aggregators shuffle the data among all processes to the desired destination. Two-phase collective I/O provides satisfying I/O performance for large scale scientific applications, and its nonblocking support is by far the best solution for overlapping the shuffle phase with the I/O phase. However, we argue that it faces challenges in extreme scale computing due to its high communication overhead and the 'two phase' constraints. To support these arguments, we profile the two phases in finer granularity. The test is a collective I/O run

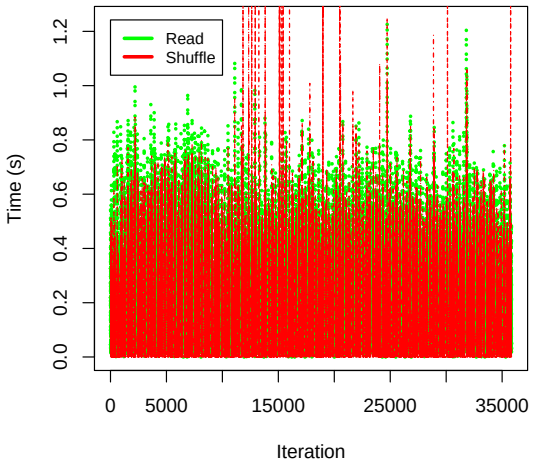

Figure 1: I/O Profiling of Two Phase Collective I/O

using 72 processes, among which 6 are aggregators on each node, and the node has 12 CPUs each. A 4-D climate dataset is distributed on 40 OSTs in a round robin fashion with 4MBs stripe size. The size of the dataset is 1024 x 1024 x 100 x 1024, from fast dimension to slowest dimension. We run the code to access a 4-D subset, which has size 100 x 100 x 10 x 720, and each process is assigned to access 100 x 100 x 10 x 10. This I/O request generates large amounts of non-contiguous small requests, therefore it can benefit from the collective I/O. Figure 1 shows the result of this I/O request with separate timing of read and shuffle phase in each iteration. (The buffer is 4MB as default). We can find that even though the shuffle phase is well overlapped by the I/O phase in a nonblocking way, the shuffle phase still consumes substantial time and the total shuffle cost approaches the actual I/O cost. By comparing the final I/O cost (including two phases), the overhead caused by the shuffle phase is about 20%. When increasing the scale, this overhead is shown to be increasing as well.

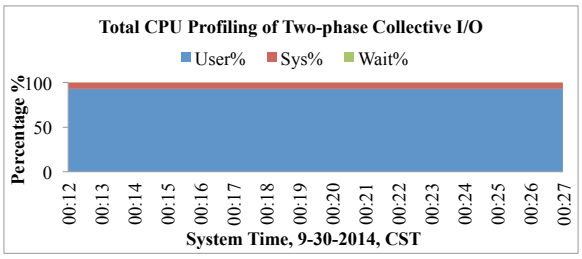

Figure 2: CPU Profiling of Two Phase Collective I/O

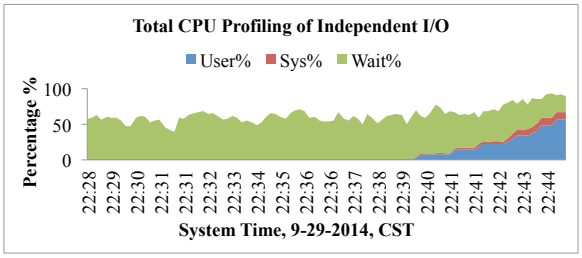

Figure 3: CPU Profiling of Independent I/O

B. MapReduce Computing Paradigm

MapReduce is a programming model that has been popular in cloud computing for a few years. Computing paradigms based on mapreduce, such as, hadoop, haloop and spark, continue to dominate today's big data market. The reason that MapReduce has become successful is its simplicity, scalability, and fault-tolerance that can be achieved for a variety of applications. Programs written in mapreduce style are parallelized automatically on a large cluster of commodity machines. MapReduce has a restricted programming framework, which also includes two phase, i.e., map and reduce. This incites us to conceptually compare the mapreduce and collective I/O. In the "Map" phase: The master node takes the computing problem as input and divides it into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node. In the "Reduce" step: The master node then collects the answers to all the sub-problems and combines them to form the output to the problem it was originally trying to solve.

III. Collective Computing

We can see that the mapreduce and two-phase collective I/O share the same idea of splitting task (computation or I/O) to enable parallelism. In this section, we propose to design a collective computing paradigm by incorporating the mapreduce workflow in the two-phase collective I/O. The fundamental idea is to break the collective I/O's 'two-phase' constraint, and insert a 'map' between the I/O, such that the analysis stage in most scientific applications are moved ahead to combine with the collective I/O into a finer-granularity non-blocking I/O (see section V for common coarse-granularity, non-blocking I/O). The collective computing framework is illustrated in Figure 4. In Figure 4, the framework contains two basic phases, I/O phase and shuffle phase, which is same with traditional two-phase collective I/O. Between the two phases, where the collective I/O used to have the data ready on compute nodes before shuffle, now we insert the 'map', such that the data is computed directly on each local node before shuffle. After the map, partial results are generated from the computation on the raw data. Therefore, the size of message to be sent is reduced dramatically. As shown in Figure 4, the shuffle phase is continued with a small amount of analysis results, and a 'reduce' process is conducted on the analysis results, which is same with mapreduce's reduce phase. This new collective computing framework essentially combines the mapreduce's reduce and collective I/O's shuffle into one phase, as shown in the figure, $p1$ has the final result.

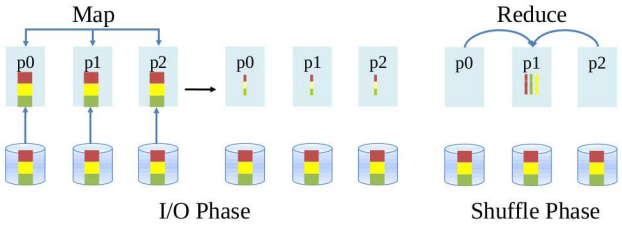

Figure 4: Collective Computing

The collective computing framework can be seen as a mapreduce-like computing paradigm for HPC based on collective I/O. Unlike the mapreduce, where I/O is just implicitly controlled and computing is explicitly parallelized by writing a map task, two-phase collective I/O is just an I/O middle-layer. It has a long way to go before it becomes an efficient computing framework, i.e., collective computing. Therefore, the first challenge is to find a way to represent the computation in the collective I/O. We introduce the concept of object I/O, where computation can be programmed into an object and explicitly passed into the collective I/O function call by users. Collective I/O is performed in byte level, but the later map operation needs to recognize the data as meaningful subsets. Therefore, When the data is ready on compute nodes after the I/O phase, we perform the map operation on each segment, then shuffle the computation results to one final compute node to compute the reduction (or distribute the results to each corresponding node). We explain the concepts and describe the design details separately in the following subsections.

A. Object I/O and Runtime Support

Traditionally, collective I/O sits between high-level APIs, e.g., PnetCDF, and low-level file system drivers, e.g., Lustre

ADIO driver. When conducting scientific analysis using the collective I/O, users usually write the parallel 'read' code first, and then write the parallel computing code using $nprocs$ processes. We illustrate a sample code in Figure 5.

```
I/O
1. start[0] = (dim/nprocs)*rank;
2. count[0] = (dim/nprocs);
3. temp=(float *)malloc(SIZE*sizeof(float));
4. ncmpi_get_vara_float_all(
   ncid, varid, start, count, temp);
Computation
5. for(i = 0;i < count[0];i++){
6. sum += temp[i];
7. }
8. MPI_Reduce(sum, SUM, size, MPI_FLOAT,
          MPI_SUM,0,MPI_COMM_WORLD);
```

Figure 5: Traditional MPI Code with Collective I/O

In Figure 5, line1 1-4 define the access region for each process, lines 5-7 specify the computation, and line 8 reduces all the sub-results into the final result. The sum operation is not ready to execute until the data in $temp$ is received. Using collective I/O, the data in $temp$ is partially accessed from the current process and partially received from other processes. In order to represent the computation into the collective I/O, we declare computation and I/O separately and group them together to comprise an object I/O. This object I/O transfers the function routine to the lower-level collective I/O call and finally reaches the two-phase layer, where the I/O aggregation and shuffling is conducted. We demonstrate this programming structure in Figure 6.

```
I/O
1. io.start[0] = (dim/nprocs)*rank;
2. io.count[0] = (dim/nprocs);
3. io.temp  = (float *)malloc(SIZE*sizeof(float));
4. io.mode  = collective;
5. io.block = false;
Computation
6. void compute(out, in, len, dtype)
7. for(i = 0;i < len;i++){
8. out += in[i];
9. }
Object
10.MPI_Op_create((MPI_User_function*)compute,1,&op);
11.ncmpi_object_get_vara_float(io,op);
```

Figure 6: Object I/O

As shown in Figure 6, the object I/O consists of three parts: I/O region, computation function, and the final object encapsulation. In the I/O region, users can specify the accessing information as usual, with the only difference in defining the I/O mode. We use ADIOS-like [16] I/O description, where I/O routines are specified separately. The $io.mode = collective$ sets the I/O as collective I/O, and other options can be $independent$. The $io.block = false$ tells the code to use our non-blocking framework, i.e., collective computing. If the $io.block$ is set to be true, then the code is essentially identical to the traditional MPI-IO code, where computation and I/O are performed separately and the two-phase collective I/O is restricted without inserting analysis code between the two phases.

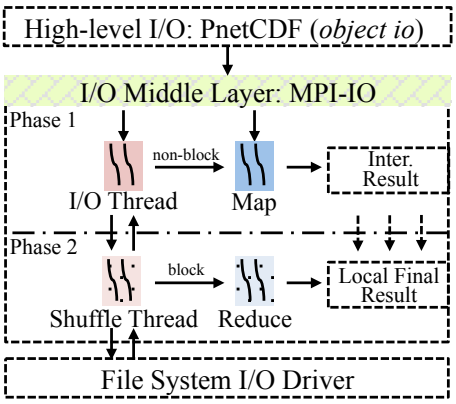

Figure 7: Collective Computing Runtime

B. Map on Logical Subsets

In collective computing, the map operation is performed on the incomplete data block. There are two reasons that the data is incomplete. One is because in the I/O phase, the data on one node is an aggregated data block, which has other processes' requested data. The aggregated block may or may not cover the local processes' requests. The second reason is that we support non-blocking collective computing, such that the data block may be only one or several iterations' return (the I/O aggregator has multiple iterations to finish the I/O, with each iteration only accessing a portion of the data).

Such an incomplete data block is just a sequence of bytes, with no self-describing metadata as in high-level I/O layer, e.g., PnetCDF. As shown in Figure 8, the high-level I/O request, e.g., HDF5 or PnetCDF, defines the logical access coordinates of the dataset. The logical information becomes lost in the MPI-IO layer. When the logical requests come into the MPI-IO layer, all processes will share their accessing information by exchanging the 'offset list'. During I/O aggregation, a global offset list is then sorted to form a contiguous access region. After dividing into several even aggregated I/Os (in Figure 8's example, it is three), the selected process, e.g., $p0$, will work as the aggregator to complete the I/O. In the second phase, when the aggregated request is fetched in the buffer, we need to reconstruct the logical coordinates of the byte sequence and then pass the logical information to the 'map' function. The construction process utilizes the offset list information and the raw dataset metadata, e.g., dimension and size. During the construction, for each byte sequence, we first calculate its corresponding offset list, e.g., $sequence_0 = \{(offset_0 = 0, length_0 = 100), ...\}$. Then we compute its coordinates in the original dataset, e.g., $sequence_0 = \{(start_0 = 0, length_0 = 10, start_1 = 0, length_1 = 10), ...\}$. Mapping

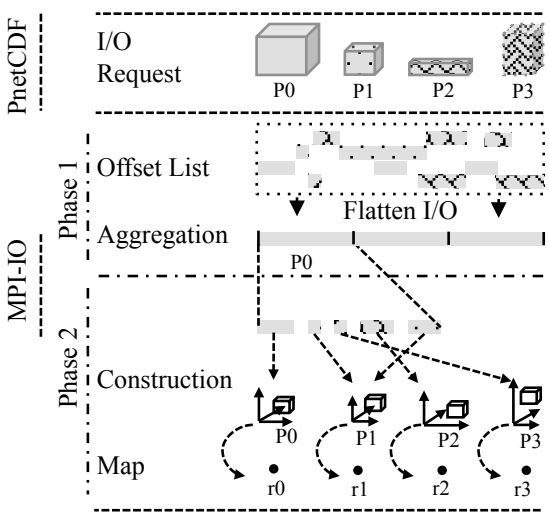

Figure 8: Map on Logical Subsets

the offset to coordinates may result in non-contiguous logical subsets (this is why $sequence_0$ can have multiple start/length pairs). We group them into one set and pass them to the 'map' function. With the logical coordinates, we are able to perform a 'map' operation on the logical subset and store the intermediate result for future 'reduce'. A map operation is a user-defined function parsed from the upper layer's object I/O. For example, the map operation can be a summation process, as shown in the sample in Figure 5. This function will go through each available logical subset in the buffer to compute the intermediate results. Therefore, a process' desired result now lies in several partial results. These intermediate partial results have the process information and logical coordinate information integrated as metadata.

C. Results Reduce and Construction

After the map operation, the intermediate results are cached in the aggregator's buffer on compute nodes. In traditional collective I/O, the contents in the buffer are supposed to be shuffled among all the processes. In collective computing, we support both all-to-one reduce and all-to-all reduce. The only difference is when the intermediate results are shuffled. In all-to-one reduce, all the intermediate results are sent to one single node, and each process' partial results are constructed on that node. The final reduce also happens on the same node. For all-to-all reduce, the intermediate results are shuffled among all processes, such that each process only has its own partial results. After this shuffle, each process conducts its own reduce locally. Then the results of each process are sent to one node to perform a final reduce. All-to-all reduce clearly brings more communication overhead than all-to-one reduce, but it is desired in some scenarios where each process has further processing on the results, locally.

IV. Experiments Results and Analyses

A. Experimental Setup

We conduct our evaluation on a cray XE6 cluster, Hopper. The Hopper cluster has 153,216 compute cores for parallel jobs, 212 terabytes of memory and 2 Petabytes of disk. Each node has two 12-core AMD 'MagnyCours' 2.1 GHz processors and at least 32 GB memory per node. The Lustre file system we used has 156 OSTs with a 35 GB/s peak I/O bandwidth. The compute nodes are connected via a custom high-bandwidth, low-latency network provided by Cray. The connectivity is in the form of a 'mesh'. The collective computing framework is implemented using MPICH 3.1.2, which is the latest stable MPICH version. We evaluate the framework with both benchmarks and real applications. The benchmark is a climate simulation with a synthetic dataset. The real application is the model used in Weather Research & Forecasting (WRF). We evaluate the collective computing in the performance, scalability, and overhead.

B. Benchmark Evaluation

We benchmark the collective computing framework using a synthetic climate dataset, which has size of 800 GBs. We simulate the computation part with different operations, e.g., sum, max, and average, etc. In the first test, we vary the ratio of computation and I/O from 10:1 to 1:10. The ratio 10:1 means computation cost is 10 times of I/O cost in the overall execution time. We would see how this computation/IO ratio can impact the collective computing performance. We run

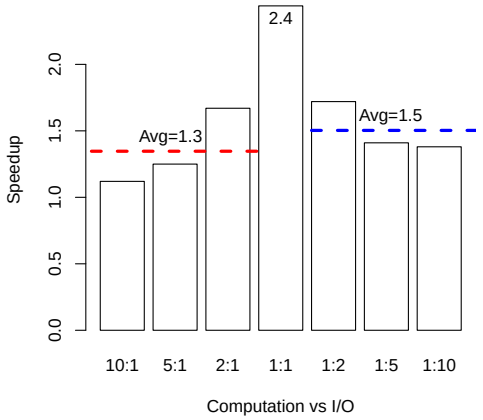

Figure 9: Speedup with Different Computation I/O Ratio

the simulation code with different ratio each time and with 120 processes. Among them, the number of aggregators is equal to the number of compute nodes, i.e., 5 (each has 24 cores). We use this configuration for aggregator as default for the following evaluations. Each test is performed three times to get the average. The I/O in each test is accessing

a 3-D subset of the climate data on only one variable, e.g., temperature. In each test, the collective computing is compared with traditional MPI computing, in which collective computing pass the computation into I/O while traditional MPI computing only perform computation after completion of the I/O. From Figure 9, the collective computing framework achieves 1.57X overall average speedup over the traditional MPI computing. Besides the overall speedup, we also observe that the different performance speedup with various computation-I/O ratio. As reported in the figure, the speedup is increased and then decreased, with the peak of speedup 2.44X at ratio of 1:1. This is because that the major benefit of the collective computing is that the analysis time is moved in advance. We also labeled two average speedup in the figure, in which, one is the average speedup of 'computation>I/O' and the other is the speedup of 'I/O>Computation'. The fact that later one is better than the former one shows that the collective computing favors the data intensive application. The reason is that the in-advance analysis transforms the large data into a small result, and this transformation benefits the communication/shuffle phase more when the data is intensive.

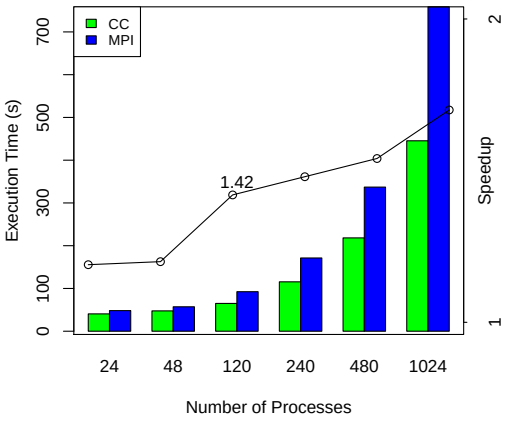

Figure 10: Scalability of Collective Computing

We continue to evaluate the collective computing on its scalability and overhead. As shown in Figure 10(a), the 'CC' stands for collective computing and 'MPI' refers to the traditional MPI computing. We set the computation-I/O ratio as 1:5, which is equal to the sixth bar in the Figure 9. We also set the request size in each process to be same, therefore, the workload is increasing monotonically with the number of processes. As the number of processes increases, with our default configuration, the number of aggregators will also increase, such that the communication/shuffle cost in traditional MPI is increased too. In this case, we find that the collective computing can further increase the performance at the fixed computation-I/O ratio, i.e., the speedup is increased from 1.42X to 1.7X as the processes increasing from 120 to 1024, which correspondingly has a reduction of execution time from 27s to 313s.

From the Section III, we can see that additional works are needed in order to support the collective computing, e.g., logical construction and intermediate results reduction (Figure 8). We sum up all the additional works as 'local reduction' overhead and plot in the Figure 11. In this test, we

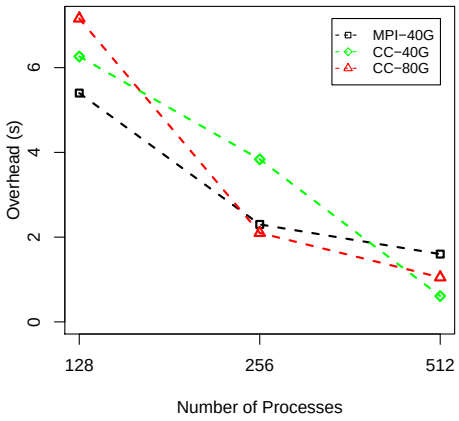

Figure 11: Overhead Analysis

run with 128, 256, 512 processes separately and measure the MPI's reduction and collective computing's local reduction. The total I/O size in each test is same, which is 40 GB or 80GB. As shown in Figure 11, we plot both collective computing and traditional MPI programming. We can see that the collective computing does not bring much overhead compared to MPI programming. The 'CC-80G' is higher than 'CC-40G' confirms that when the number of processes is the same, the workload will determine the overhead (case of #256 is an exception due to the runtime noise in the cluster, the averge case matches with our analysis). Overall, the overhead caused by collective computing's local reduction will not cause bottleneck comparing to the total I/O cost (in this experiment, the I/O cost is 76s in average).

As we see in section III-B, maintaining the logical information as metadata for the intermediate result can cause additional storage overhead. The metadata includes process information and logical coordinates. The size of the metadata are affected by many different factors. Among which, the number of aggregators, the number of processes, the MPI buffer size, and the I/O pattern are the major ones. We evaluate the storage overhead caused by the metadata. We only plot the result with varying MPI buffer size. As shown in Figure 12 The MPI buffer size has an impact on the metadata size which is similar to the typical issue in file

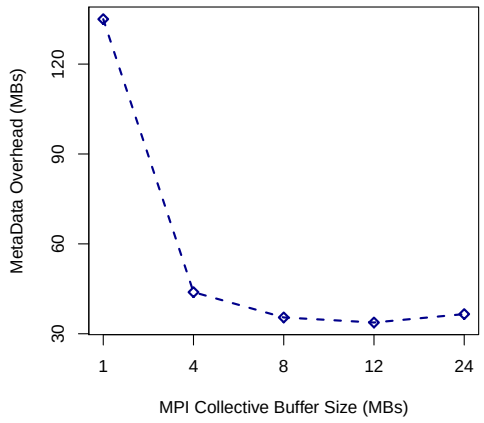

Figure 12: Metadata Overhead Analysis

system management, i.e., the size of the block has an impact on the disk space efficiency. With smaller MPI buffer size, for example 1 MB, if the intermediate logical subset has average larger size than 1MB and less than 2MB, then the subset will be broken and accessed in two iterations to fit the limited buffer size. In each of the two accesses, the collective computing runtime will generate the metadata and index the intermediate result, which eventually generates two copies of metadata for this single logical subset. But with a larger buffer size, this situation can be optimized. In Figure 12, we can find that the metadata overhead is reduced as the buffer size increases. We also find that the largest buffer size will not further reduce the overhead. Instead, the optimal buffer size is around 8 MB to 12 MB. The reason is similar to the previous mentioned typical file system problem. Besides this, the I/O pattern also has impact to the benefit of larger buffer size. For example, non-contiguous I/O for high dimensional subset often generates more smaller sub-subsets in the runtime, while contiguous I/O pattern generate more larger sub-subsets. Both cases prefer larger buffer size, but the later one benefits more from larger buffer size since smaller buffer size can also satisfy the smaller sub-subsets.

C. Application Tests

Scientific applications become data intensive and desire an efficient computing paradigm. We evaluate our collective computing framework using the Weather Research & Forecasting (WRF) [20] Model. The WRF is a next-generation mesoscale numerical weather prediction system designed for atmospheric research. The model serves a wide range of meteorological applications across scales from tens of meters to thousands of kilometers. The WRF allows researchers to generate atmospheric simulations based on real data(observations, analyses) or idealized conditions. Besides this, WRF has a large worldwide community of registered users (over 25000 in over 130 countries). Choosing WRF as our evaluation application can have a broader impact. We select two common analysis tasks found in the WRF and put in our collective computing framework. The two tasks are extracted from a hurricane simulation, which are 'Min Sea-Level Pressure (hPa)' and 'Max 10m wind speed(knots)'. This two analysis tasks have a common feature which is that the I/O is a subset access in a non-contiguous pattern and the computing is an additive operation that can be map and reduced. We plot the result of the first task in Figure 13, as the second test demonstrates similar results. As shown in Figure 13, our collective computing improves

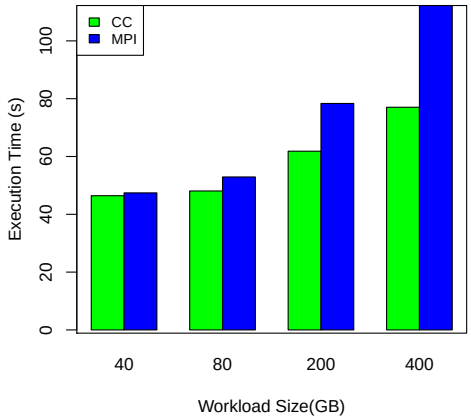

Figure 13: WRF Performance with Collective Computing

the performance of the WRF task by 1.45X speedup.

V. Related Work

A. Nonblocking Collective Operations

Nonblocking operations allow computation and communication to be overlapped and thus to leverage hardware parallelism for the asynchronous message transmission. Such a technique has existed for a decade [26]. There are two previous works in MPI that are related to our approach. They are *nonblocking collective communication*(NB-COM) [11] and *nonblocking collective I/O*(NB-CIO) [4, 24, 27]. The former standardizes the MPI's nonblocking collective operations, e.g., $MPI\_Alltoall$. When used in suitable applications, for example the three dimensional fast fourier transformations (FFT), as soon as the first data element is ready, the communication of them is started in a non-blocking way. Therefore, the communication is overlapped with the computation of the following elements. The NB-COM focuses on the MPI's process communication instead of I/O, which is different from our proposed collective computing. The NB-CIO in

current MPI based open source project, e.g., libNBC, and in the high-level I/O, e.g., PnetCDF, focuses on the I/O and has overlap with our approach, but still differs fundamentally. Existing NB-CIO supports the collective I/O's to run in a non-blocking way and overlap with computation during write and read operation. The NB-CIO is essentially a variant of NB-COM in an I/O version with only difference that NB-CIO can have different message size while NB-COM always have same message size. The two existing approaches share the non-blocking idea with our approach, but our collective computing is designed in a more finer fashion, in which the two phases are split and computation is inserted into each cycle of aggregator's I/O operation. One of our CC's functionality where computation is conducted on the same data with I/O is not supported in existing approaches. Existing approaches do support computation to overlap with I/O or communication, but the computation is actually performed on different dataset that are independent with the I/O or communication.

B. Combination of MPI and Mapreduce

MPI is a dominate programming language in high performance computing and mapreduce is a simplified computing framework in cloud computing. The combination of the two has been explored in several existing works [17]. Both of the computing frameworks have advantages and disadvantages. One example is that MPI is complex but supports flexible derived data types while MapReduce is simple but can enable embarrassingly parallel computation on a large cluster of commodity machines. Among the existing works, DataMPI [17] designs a Hadoop-like big data computing framework for HPC. This framework extends MPI to efficiently process and communicate large number of key-value pair instances in a Hadoop-like fashion. Our collective computing framework is designed for HPC, which is in an opposite direction with DataMPI. Related work towards efficient mapreduce using MPI can also be found in [9, 10, 18]. Our work differs in that we use a MapReduce-like paradigm to benefit MPI instead of the opposite way in these existing works. Besides this, we utilize the similarities between two-phase collective I/O and mapreduce and provide a flexible computing paradigm for solving scientific big data problems in current HPC.

VI. Conclusion

Scientific applications face challenges in analyzing large amounts of data sets. Traditional MPI blocking computing workflow can not conduct analysis until the I/O is finished. Existing non-blocking MPI programming also does not support computation on I/O stream in the two-phase collective I/O. We design the collective computing which is a finer non-blocking computing paradigm, that moves the analysis stage in advance. The collective computing framework breaks the two-phase I/O constraint and has an object I/O to represent computation inside the I/O phase. We design the logical map to recognize the byte sequence. We have evaluated the collective computing experimentally and have shown that it can improve the performance of scientific big data analysis by 2.5X speedup. In the future, we would like to support the iterative operations and investigate the fault tolerance of the collective computing.

References

[1] Big data meets big science. http://cacm.acm.org/magazines/2014/7/176202-big-data-meets-big-science/fulltext.

[2] B. Behzad, S. Byna, S. M. Wild, and M. Snir. Improving parallel i/o autotuning with performance modeling. In *HPDC*. 23rd International ACM Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, 2014.

[3] K. Cha and S. Maeng. Reducing communication costs in collective I/O in multi-core cluster systems with non-exclusive scheduling. *The Journal of Supercomputing*, 61(3):966–996, 2012.

[4] P. Dickens and R. Thakur. Improving collective i/o performance using threads. In *13th International and 10th Symposium on Parallel and Distributed Processing. 1999 IPPS/SPDP.*, pages 38–45, Apr 1999.

[5] P. M. Dickens and R. Thakur. Evaluation of collective I/O implementations on parallel architectures. *Journal of Parallel and Distributed Computing*, 61(8):1052–1076, Aug. 2001.

[6] M. Folk, A. Cheng, and K. Yates. HDF5: A file format and i/o library for high performance computing applications. In *Proceedings of Supercomputing'99 (CD-ROM)*, Portland, OR, Nov. 1999. ACM SIGARCH and IEEE.

[7] K. Gao, W. keng Liao, A. N. Choudhary, R. B. Ross, and R. Latham. Combining I/O operations for multiple array variables in parallel netCDF. In *CLUSTER*, pages 1–10. IEEE, 2009.

[8] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. Scientific and engineering computation. MIT Press, pub-MIT:adr, 2000.

[9] Y.-F. Ho, S.-W. Chen, C.-Y. Chen, Y.-C. Hsu, and P. Liu. A mapreduce programming framework using message passing. In *Computer Symposium (ICS), 2010 International*, pages 883–888, Dec 2010.

[10] T. Hoefler, A. Lumsdaine, and J. Dongarra. Towards efficient mapreduce using mpi. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 240–249, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking

Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[12] C. Jin, S. Sehrish, W. keng Liao, A. N. Choudhary, and K. Schuchardt. Improving the average response time in collective I/O. In *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, volume 6960 of *Lecture Notes in Computer Science*, pages 71–80. Springer, 2011.

[13] W. keng Liao and A. Choudhary. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In *SC'08*. ACM/IEEE, Austin, TX, Nov. 2008.

[14] Q. Koziol. HDF5. In *Encyclopedia of Parallel Computing*, pages 827–833. Springer, 2011.

[15] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: a high-performance scientific I/O interface. In *SC2003*, 2003.

[16] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich io methods for portable high performance io. In *Proceedings of IPDPS'09, May 25-29, Rome, Italy*, 2009.

[17] 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,IPDPS'14*, pages 829–838, May 2014.

[18] H. Mohamed and S. Marchand-Maillet. Enhancing mapreduce using mpi and an optimized data exchange policy. In *41st International Conference onParallel Processing Workshops (ICPPW),2012*, pages 11–18, Sept 2012.

[19] R. Ross, R. Latham, M. Unangst, and B. Welch. Parallel I/O in practice, tutorial notes. In *SC'08*. ACM/IEEE, Austin, TX, Nov. 2008.

[20] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers. A description of the advanced research wrf version 2. *AVAILABLE FROM NCAR; P.O. BOX 3000; BOULDER, CO*, 88:7–25, 2001.

[21] Y. Su and G. Agrawal. Supporting user-defined subsetting and aggregation over parallel netCDF datasets. In *CCGRID*, pages 212–219. IEEE, 2012.

[22] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE, Feb. 1999.

[23] Y. Tian, S. Klasky, H. Abbasi, J. F. Lofstead, R. W. Grout, N. Podhorszki, Q. Liu, Y. Wang, and W. Yu. EDO: Improving read performance for scientific applications through elastic data organization. In *CLUSTER*, pages 93–102. IEEE, 2011.

[24] V. Venkatesan, M. Chaarawi, E. Gabriel, and T. Hoefler. Design and Evaluation of Nonblocking Collective I/O Operations. In *Recent Advances in the Message Passing Interface (EuroMPI'11)*, volume 6960, pages 90–98. Springer, Sep. 2011.

[25] Y. Wang, Y. Su, and G. Agrawal. Supporting a light-weight data management layer over HDF5. In *CCGrid*, pages 335–342. IEEE Computer Society, 2013.

[26] Non-blocking algorithm. http://en.wikipedia.org/wiki/Non-blocking_algorithm.

[27] Y. Yu, J. Wu, Z. Lan, D. Rudd, N. Gnedin, and A. Kravtsov. A transparent collective i/o implementation. In *IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS), 2013*, pages 297–307, May 2013.

[28] X. Zhang, K. Davis, and S. Jiang. IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination. In *SC'10*. ACM/IEEE, New Orleans, LA, USA, Nov. 2010.