Decoupled I/O for Data-Intensive High Performance Computing

Chao Chen*, Yong Chen*, Kun Feng†, Yanlong Yin†, Hassan Eslami‡
Rajeev Thakur§, Xian-He Sun†, William D. Gropp‡
* *Department of Computer Science, Texas Tech University*
† *Department of Computer Science, Illinoise Institude of Technology*
‡ *Department of Computer Science, University of Illinois Urbana-Champaign*
§*Mathematics and Computer Science Division, Argonne National Laboratory*
{*chao.chen, yong.chen*}*@ttu.edu, kfeng1@hawk.iit.edu, yyin2@iit.edu, eslami2@illinois.edu*
thakur@mcs.anl.gov, sun@iit.edu, wgropp@illinois.edu

***Abstract*—The I/O bottleneck issue has been acknowledged as one of main performance issues of high performance computing (HPC) systems for data-intensive scientific applications, and has attracted intensive studies in recent years. With the enlarging gap between the computing bandwidth and I/O bandwidth in projected next-generation HPC systems, this issue will become even worse. In this paper, we present a novel decoupled I/O to address the fundamental I/O bottleneck issue. The decoupled I/O is a software stack including MPI extensions, compiler improvements, and runtime library support, based on a decoupled HPC system architecture. It allows users to treat the computing of data-intensive operations and the traditional I/O operation as an ensemble and offload them into dedicated data nodes, which are near to the data source, to reduce the overhead of data movement and improve the I/O bandwidth usage. The decoupled I/O is user-friendly and requires little changes in application codes. Experiments were conducted to evaluate the performance of the decoupled I/O, and the results show that it outperforms existing solutions (such as active storage I/O) and provides an attractive I/O solution for data-intensive high performance computing.**

***Keywords*-Decoupled I/O, data-intensive computing, high performance computing, parallel I/O, storage**

I. Introduction

Many scientific applications nowadays tend to be highly data-intensive. These applications are intended to read/write terabytes or even petabytes of data in a single simulation run. For instance, the output volume of the Arctic Systems Reanalysis (ASR), which aims to study the climate change during the period from 2000 to 2010 (11 years), is nearly 23.14TB [19]. To achieve better understandings of scientific phenomenon, the data volume generated by scientific simulations keeps increasing year by year because of finer resolution requirements, and is projected to be around hundreds of petabytes for a single simulation run on the coming exascale systems (projected to be implemented by 2018∼2020). Compared with existing petascale systems, the computational power of the exascale systems will be improved by 1000 times with exploring billion-way concurrency (with $O(1M)$ nodes and $O(1K)$ cores within a node), whereas *the I/O bandwidth will only be improved around 10-30 times*. This increasing performance gap between the computing bandwidth and I/O bandwidth has made the I/O subsystem become a critical bottleneck of HPC systems. Moving hundreds of petabytes datasets among storage nodes and compute nodes is not wise and will seriously limit the application performance in the exascale era. It is also not energy efficient because of the overhead of data movements, whereas a fixed power consumption envelope is another critical design factor for exascale systems. Reducing the data movement unarguably is a critical challenge to be addressed for the coming exascale I/O systems.

Previous studies have shown that moving computations near to the data is a promising solution for I/O bottleneck issue. It can efficiently reduce the data movement and the power consumption, through offloading data-intensive operations near to data and transferring reduced results between storage nodes and compute nodes. Active storage, Fast-Forward I/O (including burst buffer) and active disk/flash are among the most important approaches, and attracting intensive studies.

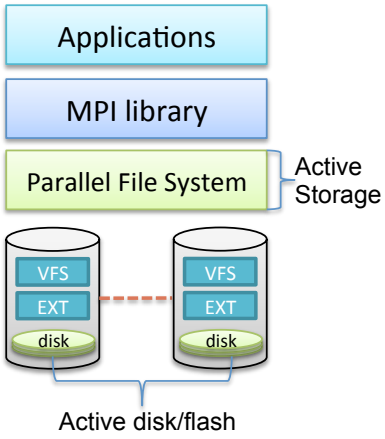

Figure 1: I/O Software Stack of High Performance Computing Systems

In this paper, we study a new way of moving computations near to data to minimize the data movement with a *decoupled I/O* approach to address the I/O bottleneck issue for

data-intensive applications. Decoupled I/O is a runtime system designed for our previous decoupled execution paradigm (DEP) proposed in [6]. It allows users to combine the computation from data-intensive operations and traditional I/O operations as an ensemble and offload them to the data nodes. A user-friendly interface is provided for programmers to port existing applications to the decoupled I/O system without significant code changes. The programmers can flexibly implement and offload codes to the data nodes by implementing them as offloaded objects. The current prototype and evaluations have shown a promising potential.

The rest of paper is organized as follows. Section II discusses background and motivations of this research. Section III and Section IV present the design of the decoupled I/O and its evaluations respectively. Section V discusses the existing related work, and Section VI concludes this study.

II. Background

The decoupled execution paradigm (DEP), introduced in our previous work [6], can address the resource contention issue by using dedicated data nodes for data processing. The architecture of DEP is illustrated in Figure 2. This new architecture decouples nodes into compute nodes and data processing nodes (also called data nodes). These nodes are designed to be mapped with computation-intensive operations and data-intensive operations respectively. Computation-intensive operations are executed on massive compute nodes, while data-intensive operations are executed on dedicated data nodes. Physically, the data nodes are further divided into compute-side data nodes and storage-side data nodes. They are closely connected (for example, with direct links or high-speed interconnection switches) to data sources through a high-speed network.

One critical challenge involved in the DEP architecture is to decouple the data-intensive operations (codes) of an application to the dedicated data nodes without requiring significant changes for programmers. In this research, we propose a decoupled I/O to help users to easily program for DEP architecture. Currently, the proposed solution in this paper mainly focuses on addressing storage-side data nodes. The decoupled I/O provides programmers an easy way to offload data-intensive operations to minimize data movements through a set of APIs. The next section details the design and implementation of the proposed decoupled I/O.

III. Decoupled I/O

A. Overview of Decoupled I/O

The decoupled I/O is designed as an extension of the existing MPI library. It improves the MPI library to manage the compute nodes and data nodes as an ensemble, as shown in Figure 3. The decoupled I/O, however, internally splits them into two different groups: compute group and data group. The compute group runs the normal applications,

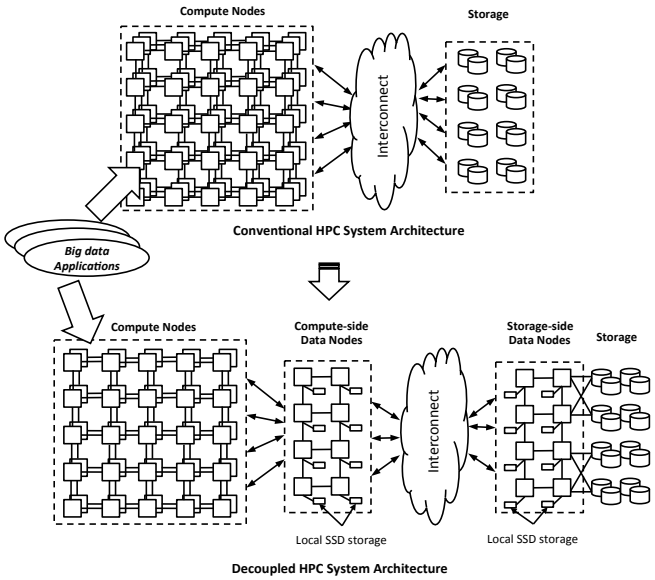

Figure 2: Decoupled High Performance Computing System Architecture

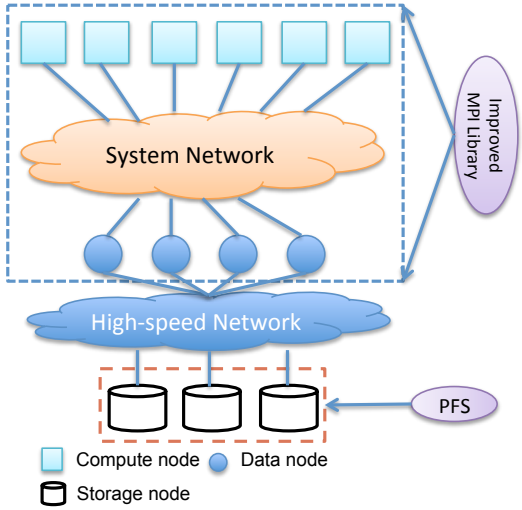

Figure 3: A High-level View of Decoupled I/O

while the data group executes the offloaded data-intensive operations. Both of them share the same MPI library.

For programmers, the decoupled I/O provides a user-friendly interface to enable them to offload data-intensive operations to minimize data movements. They can use the designed APIs to migrate their codes to a decoupled execution platform without significant changes. Table I illustrates the differences between a traditional code and the decoupled I/O code through an example. The only change for the decoupled I/O code is that the programmer needs to implement the code as a function and invoke it through a provided API.

The decoupled I/O involves three improvements to exist-

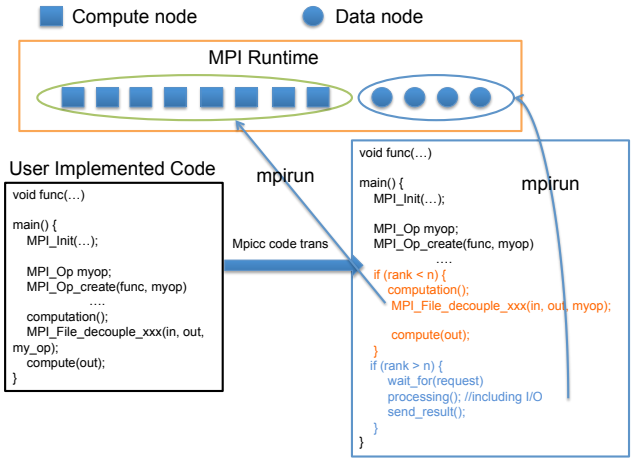

Figure 4: Decoupled I/O at Runtime

Table I. Comparison between a Traditional Code and a Decoupled I/O Code

```
/* Traditional Code */
... int buf;
MPI_File_read(fh, buf, ...);
for(i = 0; i < bufsize; i++) {
  sum += buf[i];
}
```

```
/* the code using decoupled I/O, user implement the operation */
int sum_op(buf, bufsize) {
  for (i = 0; i < bufsize; i++ )
    sum += buf[i];
}
....
MPI_op myop;
MPI_Op_create(myop, sum_op);
MPI_File_decoupled_read(fh, sum, myop, ....);
....
```

ing MPI library. Despite a set of decoupled I/O APIs, which is designed as an extension to the MPI-IO library, the MPI compiler (*mpicc*) is also improved to automatically generate the code for data processes. These processes are completely hidden from users except for having hints that can be supplied by users. In addition, the MPI process manager (hydra) is improved to manage the compute processes and data processes. Figure 4 demonstrates how the decoupled I/O works. The system manages the compute nodes and data nodes as two groups, with each group conducting different tasks. The compute group conducts the computation of applications, and the data group conducts data processing for reducing data movements. From the view of programmers, the decoupled I/O provides a set of I/O APIs to combine the data and computation together, and instruct data-intensive operations to be offloaded to data nodes. The new APIs do not involve any changes to programming logic. In addition, there is no change for programmers to construct the code architecture, except enclosing the offloaded code with a function (an offloaded object) and instructing that. The users only need to use additional arguments for *mpirun/mpiexec* to invoke appropriate processes on compute/data nodes. For instance, a runtime command *$ mpirun -np 8 -dp 4 -f hostfile ./app* instructs to launch the application with 8 processes in total with 4 processes used as data processing processes. The decoupled I/O is designed and implemented with a minimal intervention from programmers/users to minimize data movements.

B. API Design and Implementation

In general, a data-intensive operation can be considered with two phases including a traditional I/O operation and a data processing operation. Applications retrieve the data from storage and then apply the processing operations on the data to get the final results for further computations (or generate a buffer of data using a simulation kernel and then write the data to storage). The decoupled I/O treats these two phases as an ensemble that would be decoupled and executed at data nodes to significantly reduce the overheads of data movement and improve the I/O bandwidth usage for applications. For the convenience of programmers, a set of decoupled I/O APIs are provided as shown in Table II.

As the name indicates, a decoupled I/O API, $MPI\_File\_decouple\_xxx$, completes the same I/O operation as implied by MPI_File_xxx defined in the MPI standard. The decoupled I/O API, however, is associated with an "offloaded object" that is decoupled from the original code to be executed on the data nodes. For example, $MPI\_File\_decouple\_open$ will open a file for following decoupled I/O operation, and $MPI\_File\_decoupled\_read$ will read a portion ($count$) of data with type $data\_type$ from the file and then process it with assigned operation ($data\_op$) on data nodes, with returning the processed results (filled in buf) to the compute nodes. Similarly, $MPI\_File\_decouple\_write$ will execute the operation $data\_op$ on the data nodes to generate a set of data, and then write to storage (from data nodes to storage, instead of from compute nodes to storage in a traditional case). The buf is used to store any initial parameters required by $data\_op$ operation.

With the decoupled I/O, data access requests from compute processes are collected and sent to the data nodes. All of the actual I/O operations are carried out on data nodes. This mechanism is similar to a client-server remote-procedure call (RPC) framework. Figure 5 depicts the details of this mechanism. The request from each compute process is defined as a task in Table III. These tasks from the same API will be gathered together at a master process of the compute group, and then sent to a master process of the data group. The master process of the data group will distribute the tasks to each member, collect the processed results and return them to the compute group.

Table II. Decoupled I/O APIs

MPI_File_decouple_open(MPI_Decoupled_File fh, char * filename, MPI_Comm comm);
MPI_File_decouple_close(MPI_Decoupled_File fh, MPI_Comm comm);
MPI_File_decouple_read (MPI_Decoupled_File fh, void *buf, int count, MPI_Datatype data_type, MPI_Op data_op, MPI_Comm comm);
MPI_File_decouple_write(MPI_Decoupled_File fh, void *buf, int count, MPI_Datatype data_type, MPI_Op data_op, MPI_Comm comm);
MPI_File_decouple_set_view(MPI_Decoupled_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char * datarep, MPI_Info info, MPI_Comm comm);
MPI_File_decouple_seek(MPI_Decoupled_File fh, MPI_Offset offset, int whence, MPI_Comm comm);

Table III. Decoupled Task

```
typedef struct {
MPI_Decoupled_File fh; // file handle
int rank;          // Process rank in compute group
enum {READ, WRITE, READ_ALL, WRITE_ALL} io_op;
MPI_Op op;         //decoupled operation
MPI_Datatype datatype;  //data type of each element
int count;         // data size of each operation
} Decouple_task;
```

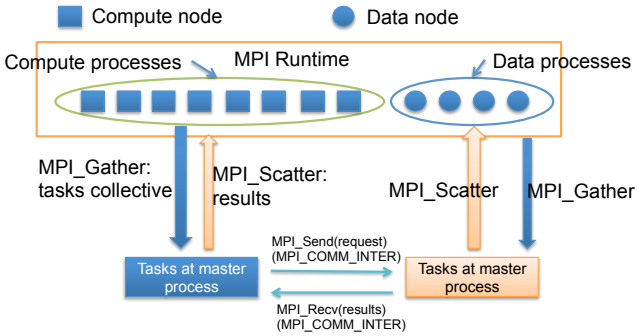

Figure 5: Decoupled I/O Implementation

C. Process/Node Management

Instead of a hierarchical structure design that separates the compute nodes and data nodes at different levels, the decoupled I/O improves the MPI library to manage the data nodes and compute nodes at the same level with two groups. When a user launches an application with a command like "mpirun -np n -dp m -f hostfile ./app", the Process Manager will automatically invoke n compute processes and m data processes, on compute nodes and data nodes, respectively. The size of each group (compute_group_size and data_group_size) will be exported to the MPI runtime. In our prototype, all of the processes, including compute processes and data processes, belong to the MPI_COMM_WORLD communicator with distinguished rank (we call world rank). At the same time, each process has its own group communicator MPI_COMM_LOCAL as an intra-communicator, and an MPI_COMM_INTER communicator as a group-to-group inter-communicator between the compute processes group and data processes group. All of these works are done at the MPI_Init stage.

D. Code Decoupling & Compiler Improvement

How to offload the user-implemented codes to data nodes is another challenge for the decoupled I/O, as our system is a distributed memory system and the address space is separated on different nodes. In this study, we introduce an easy-to-use method to address this issue. The proposed method relies on both Process Manager and MPI compiler to complete the decoupled operations. As both compute processes and data processes belong to the same global communicator: MPI_COMM_WORLD, the world rank of each process is used to identify the process type (compute process or data process) to execute different codes. This design is similar to a master-slave processing framework, where the rank is used to assign different tasks to each process when MPI applications are coded. In the decoupled I/O, the data processing code is automatically transformed by the enhanced MPI compiler, and users only need to write compute process code. Table IV illustrates an example of the user implemented code and the code translated by the compiler. In generating the code, the compiler relies on hints from users to identify the codes that run on compute nodes. The code between $MPI\_DECOUPLE\_START$ and $MPI\_DECOUPLE\_END$ is defined to be the compute process code. MPI_Op is used to define user offloaded functions. All of these operations have to be registered at the MPI runtime before $MPI\_DECOUPLE\_START$ macro is executed.

The data processes running on data nodes take care of I/O operations collectively, including opening a file and reading or writing data. To keep the logic view of compute processes as the way programmers expect, the data processes create a file handle and a file descriptor for each compute process (they are also required to open a file). Both file handle and the process rank of its local group are used to index the true file descriptor as shown in Figure 6.

IV. Evaluation

To evaluate the potential of the decoupled I/O, a set of experiments was conducted on two computing platforms. Three schemes, including the decoupled I/O (denoted as DEPIO), the state-of-the-art active storage I/O (denoted as AS), and a traditional storage I/O (without decoupled I/O or active storage I/O, denoted as TS), were evaluated in our experiments. In the decoupled I/O scheme, a set of data nodes are dedicated for serving data-intensive operations. In

Table IV. Decoupled I/O Code Sample

```
/* User Implemented Code: main.c */

int sum(int buf[], int bufsize) {
// user implemented data-intensive code
    ...
}
int main()
{
  MPI_Init(&argc, &argv);
  MPI_op myop;
  MPI_Op_create(myop, sum);
  /* offloaded operations have to be
   * registered before this macro
   */
  MPI_DECOUPLE_START;
  MPI_Comm_rank(MPI_COMM_LOCAL, &rank);
  MPI_Comm_size(MPI_COMM_LOCAL, &size);
  ......
  MPI_File_decoupled_read(fh, ...);
  ......
  MPI_DECOUPLE_END;
  MPI_Finalize();
  return 0;
}
```

```
/* MPICC translated code: main.c */

int sum(int buf[], int bufsize) {
  ......
}
int main()
{
  MPI_Init(&argc, &argv);
  MPI_op myop;
  MPI_Op_create(myop, sum);
  if (w_rank < compute_group_size) {
    /* computation code */
    MPI_Comm_rank(MPI_COMM_LOCAL, &rank);
    MPI_Comm_size(MPI_COMM_LOCAL, &size);
    ......
    MPI_File_decoupled_read(fh, ...);
    ......
  } else {  // code for data processes
    MPI_Comm_rank(MPI_COMM_LOCAL, &rank);
    MPI_Comm_size(MPI_COMM_LOCAL, &size);
    while(1) {
      if (rank == 0) {
          /* listen & distribute the
           * requests to each member */
      }
      switch(io_op) {
        case READ:
          MPI_File_read(fh, ...);
          my_op->func();
          break;
      case WRITE:
          my_op->func();
          MPI_File_write(fh, ...);
          break;
          ......
      } } }
  MPI_Finalize();
  return 0;
}
```

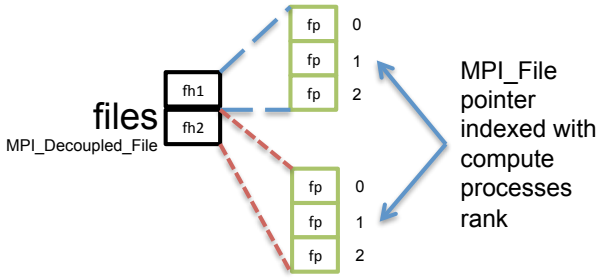

Figure 6: File Descriptors and File Pointers Management in Data Processes

the active storage scheme, the data-intensive operations were executed on storage nodes. In this section, we present our evaluation platform and results.

A. Experimental Platform

Experimental evaluations were performed on the DISCFarm (Data-Intensive Scalable Computing Farm) cluster and Hrothgar Cluater at Texas Tech University. Prototyping evaluations were performed on the DISCFarm cluster, and a large-scale emulation evaluation was performed on the Hrothager cluster (as the Hrothager cluster is a production system and we couldn't deploy the entire decoupled I/O system for evaluations).

The DISCFarm cluster is composed of one Dell PowerEdge R515 rack server node and 15 Dell PowerEdge R415 nodes, with a total of 32 processors and 128 cores. In our prototyping evaluations, 2 nodes were configured as PVFS2 storage nodes, 4 nodes were configured as data nodes, and 8 nodes were configured as compute nodes. The Hrothgar cluster consists of 640 nodes, and each node is equipped with Intel Xeon 2.8GHz CPUs (12 cores per node) and 24GB memory. It has 600TB Lustre parallel file system storage.

B. Evaluated Operations

Four operations, including data assimilation, a flow-routing operation, summation, and lookup, were used to evaluate and compare the decoupled I/O, state-of-the-art active storage I/O, and the traditional I/O.

Data assimilation is a widely used operation in climate sciences. The climate scientists use it to generate reliable climate data sets. This operation needs to read and write a large volume of data from or to storage systems. Several assimilation algorithms are commonly used, and the Ensemble Kalman Filter (EnKF) [12] is one of notable algorithms. In this study, the EnKf data assimilation was used for evaluations. It consists of 6 matrix multiplications, 1 matrix addition, and 1 matrix substitution on data retrieved [12]. The flow-routing operation is a commonly used operation to compute the direction where fluids flow to, such as in dynamics fluids simulations and geographical information systems (GIS). The summation (SUM) and lookup operations are simpler as compared with the data assimilation and

flow-routing operations. Both of them are widely used in climate data analysis applications and many other scientific computing applications. A SUM operation calculates the total value of all specified data elements, and the lookup operation searches for and returns all elements that meet given criteria from a large volume of data sets.

C. Performance of the Decoupled I/O

In this set of tests, we evaluated the performance of the decoupled I/O against the state-of-the-art active storage I/O. Both of these two schemes were configured with 2 storage nodes. To keep the same computation resources, the decoupled I/O was configured with 4 data nodes and 8 compute nodes, while the active storage I/O was configured with 12 compute nodes. The EnKF operation was evaluated. Its required computations are dominated by 6 matrix multiplications (the entire operation consists of 6 matrix multiplications, 1 matrix addition, and 1 matrix substitution).

Figure 9 shows that the decoupled I/O achieved better performance than the active storage I/O consistently. This performance achievement mainly comes from the decoupled I/O associated with data nodes, which provides more computing power for offloaded operations than that provided by storage nodes. Figure 10 further illustrates the advantage of the decoupled I/O by comparing the performance improvement against the active storage I/O under different CPU usages of each storage node. The decoupled I/O achieved performance improvement under different CPU usages, while the active storage I/O performed even worse than the traditional storage I/O for these cases.

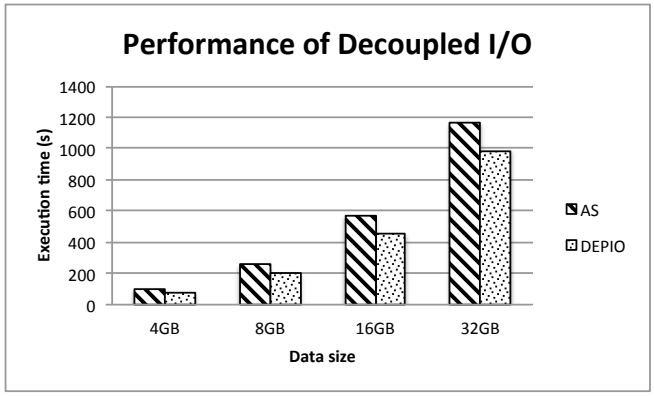

Figure 7: Performance Comparison of Decoupled I/O and Active Storage I/O (Observed CPU usage: 1.3%)

The benefits of the decoupled I/O came from two aspects: more computational and balanced resources for offloaded data-intensive operations than the active storage I/O, and also less data movements and network congestion than the traditional storage I/O. If the interconnection among data nodes and storage nodes in our test bed was high-speed networks such as InfiniBand or Myrinet, the performance of the decoupled I/O is expected to be even better.

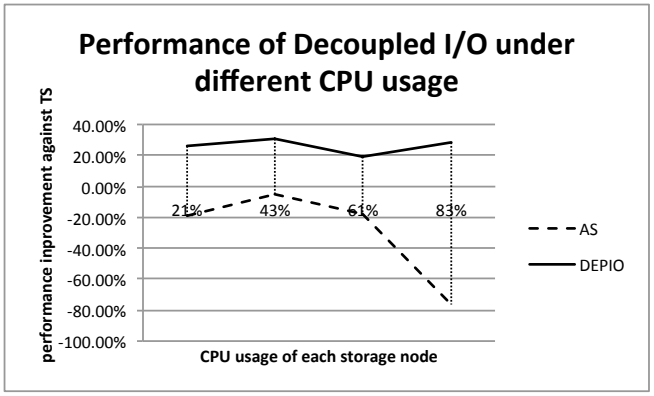

Figure 8: Performance of Decoupled I/O under Different CPU Usages

D. Emulation Results and Analysis

In this subsection, we present emulation experiments conducted on the production 640-node Hrothgar cluster to evaluate the performance of decoupled I/O for large-scale computing by comparing it with the traditional storage I/O. The observed performance improvement pattern was similar for various operations and we present the performance result of the flow-routing operation evaluated. In this set of tests, 60GB data is processed. As shown in Figure 11, the decoupled I/O was able to reduce around 25% execution time for the flow-routing operation. Figure 12 plots the achieved I/O bandwidth for the decoupled I/O and traditional storage I/O. Similarly, compared with traditional storage system, there was around 23% improvement for the decoupled I/O. All of these achievements come from the decoupled I/O associated with data nodes for data-intensive operations with reduced data movements.

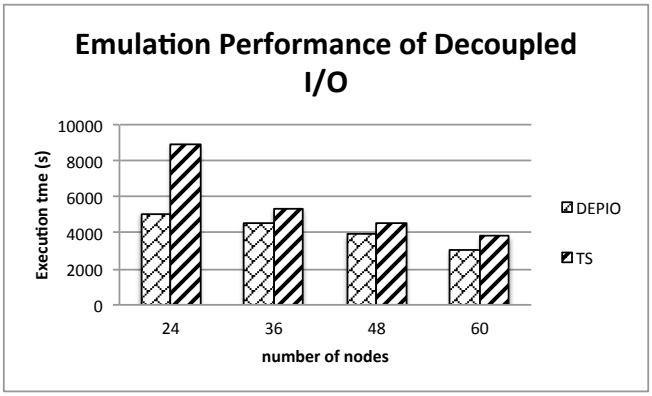

Figure 9: Emulation Performance of the Decoupled I/O

E. Overhead of the Decoupled I/O

As mentioned in previous sections, when data processes conduct the decoupled I/O, all of the decoupled tasks are gathered by a master process in the compute group, and then sent to the master process of the data group. After that,

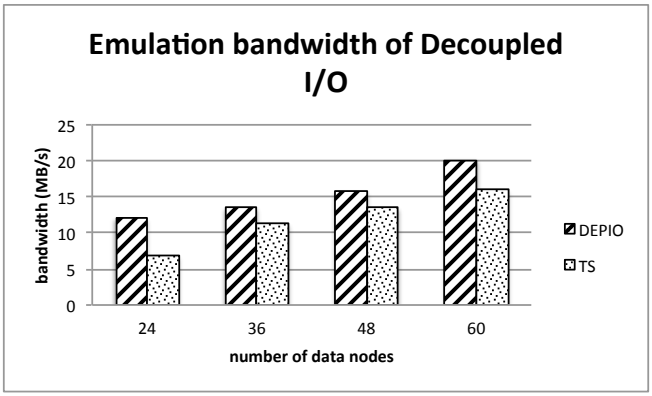

Figure 10: Bandwidth of the Decoupled I/O

the master process of data group will further distribute the tasks to its members, and then collect the results and return them back to compute processes. The primary overhead of the decoupled I/O is the communication involved in this process. In this subsection, we focus on evaluating this overhead of the decoupled I/O. In this experiment, we executed 60 compute process and 6 data processes. Figure 13 plots the result with the SUM operation evaluated, which depicts that the communication overhead is limited to a small portion of the total execution time. As the data size of each I/O request increases, this overhead will decrease steadily. Therefore, the communication overhead is unlikely to become the bottleneck in our design.

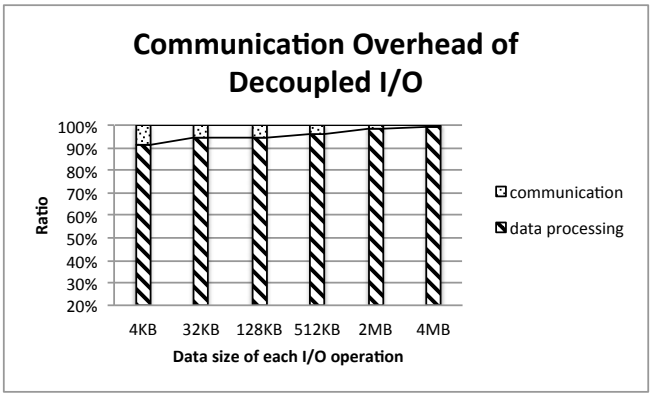

Figure 11: Overhead of the Decoupled I/O Operation

V. Related Work

High performance I/O has attracted intensive attentions and research studies in recent years. In this section, we briefly review notable solutions and compare with our study.

Active disk and active storage are two of the most notable approaches that move computations near to data to improve the I/O performance and have gained intensive attentions. Active disk [2, 9, 15, 23] reduces the data movement between the disk and memory by utilizing the computing power of the embedded CPU on the disks. *Riedel et. al.* [23] presented a detailed analysis of active disks for scan-intensive applications. However, the limited computation of embedded processor makes it difficult to have significant impact on petascale data-intensive applications. Active storage [10, 22, 24, 25, 30] shares a similar idea with active disk. It explores the computation resources of storage nodes. *Felix et. al.*[10] presented the first real implementation of active storage on the *Lustre* file system. *Woo et. al.* [29] improved the design for *Parallel Virtual File System* (*PVFS*). These studies have clearly presented the promise of approaches that move computation near to data. However, all these systems were designed without considering the resource contention issue [5]. The recent Blue Gene active storage [11] is introduced with dedicated data nodes. This study is an improvement to Blue Gene active storage. We proposed an easy-to-use software stack to flexibly utilize computing resources of both data nodes and compute nodes.

There also have been significant amount of research efforts in optimizing data-access performance using runtime libraries, such as collective I/O [8, 14, 26], data sieving, server-directed I/O, disk-directed I/O, lightweight I/O [20], partitioned collective I/O [31], layout-aware collective I/O [8], ADIOS library [16], and resonant I/O [32]. These strategies collect and aggregate small requests into larger ones at the I/O client/middleware/server level. This study naturally complements these solutions. With the decoupled I/O, these strategies can be leveraged on data nodes, and the decoupled I/O advantages are additions.

Many caching, buffering, staging, and prefetching optimization strategies exist at runtime as well, such as collective caching [13], collective buffering [18], active buffering [17], discretionary caching [28], SpecHint prefetching [4], transparent informed prefetching (TIP) [21], adaptive prefetching based on time series modeling [27], multiple-level caching and prefetching for Blue Gene systems [3]. Abbasi et. al. proposed a DataStager framework with data staging services that move output data to dedicated staging or I/O nodes prior to storage, which has been proven effective in reducing the I/O overheads and interference on compute nodes [1]. Zheng et. al. proposed a preparatory data analytics (PreDatA) to prepare and characterizing scientific data when generated (e.g. data reorganization and metadata annotation) to speedup subsequent data access [33]. These approaches have shown considerable performance improvement with dedicated output staging services and preparatory analysis. A decoupled I/O studied in this research leverages dedicated nodes as well, but is different in the sense that dedicated data processing nodes work for both reads and writes. These data nodes can provide buffering or staging too, but more importantly on data reduction. The notion of the decoupled I/O and data processing nodes is a rethinking of HPC system to provide balanced computational and data-access capability. The decoupled I/O considers to address the fundamental data movement bottleneck issue for data-

intensive applications.

VI. Conclusion and Future Work

Undoubtedly, scientific discoveries and innovations can benefit considerably from large-volume data sets generated or analyzed by scientific applications. However, the large-volume data sets have also brought up an important question with regard to data movements to the high performance computing research and development community. The massive amount of data movement and long access delay for accessing these data sets can significantly limit the productivity of data-intensive sciences.

In this study, we present a new high performance I/O solution, named decouple I/O, to reduce the data movement and improve the I/O bandwidth usage for data-intensive applications. The decoupled I/O provides a set of APIs for programmers to map computation-intensive and data-intensive operations to conventional compute nodes and dedicated data nodes respectively. We have conducted the prototyping evaluations and emulations on a production HPC system to verify the idea and the potential. The results, compared to the state-of-the-art active storage I/O, have confirmed a promising potential of such a decoupled I/O system.

While this study is one step to build better HPC systems for data-intensive applications, the current results are encouraging. The current study confirms that a decoupled I/O has its potential and usefulness in reducing data movements for data-intensive applications. With the growing importance of supporting data-intensive sciences and big data applications, the decoupled I/O can have an impact. It can be potentially built in the next-generation exascale HPC systems to better support data-intensive sciences. In the near future, we plan to continue the research investigation and to integrate the design and development of a MapReduce-style data processing into the decoupled I/O system.

VII. Acknowledgment

This research is sponsored in part by the National Science Foundation under grants CNS-1162540, CNS-1162488, and CNS-1161507. The authors also acknowledge the High Performance Computing Center (HPCC) at Texas Tech University at Lubbock for providing HPC resources that have contributed to the research results reported in this paper. URL: http://www.hpcc.ttu.edu.

References

[1] H. Abbasi, M. Wolf, and G. e. Eisenhauer. DataStager: Scalable Data Staging Services for Petascale Applications. In *HPDC*, 2009.

[2] A. Acharya and J. Saltz. Active Disks : Programming Model , Algorithms and Evaluation. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, 1998.

[3] J. G. Blas, F. Isailă, J. Carretero, R. Latham, and R. Ross. Multiple-Level MPI File Write-Back and Prefetching for Blue Gene Systems. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 164–173, 2009.

[4] F. Chang and G. A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the 3rd Symposium on Operating SystemsDesign and Implementation*, OSDI '99, pages 1–14, 1999.

[5] C. Chen, Y. Chen, and P. C. Roth. DOSAS: Mitigating the Resource Contention in Active Storage Systems. In *In the Proc. of IEEE International Conference on Cluster Computing 2012 (Cluster'12)*, 2012.

[6] Y. Chen, C. Chen, X.-H. Sun, W. D. Gropp, and R. Thakur. A decoupled execution paradigm for data-intensive high-end computing. In *In the Proc. of the IEEE International Conference on Cluster Computing 2012 (Cluster'12)*, 2012.

[7] Y. Chen, C. Chen, Y. Yin, X.-H. Sun, R. Thakur, and W. D. Gropp. Rethinking high performance computing system architecture for big data applications. Technical Report, 2013.

[8] Y. Chen, X.-H. Sun, R. Thakur, P. Roth, and W. Gropp. LACIO: A New Collective I/O Strategy for Parallel I/O Systems. In *2011 IEEE International Parallel Distributed Processing Symposium*, 2011.

[9] G. Chockler and D. Malkhi. Active Disk Paxos with infinitely many processes. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[10] E. J. Felix, K. Fox, K. Regimbal, and J. Nieplocha. Active Storage Processing in a Parallel File System. In *6th LCI International Conference on Linux Clusters: The HPC Revolution*, 2005.

[11] B. G. Fitch, A. Rayshubskiy, M. P. T.J. Chris Ward, B. Metzler, H. J. Schick, B. Krill, P. Morjan, and R. S. Germain. Blue Gene Active Storage. In *HEC FSIO R&D Workshop '10*, 2010.

[12] P. L. Houtekamer, H. L. Mitchell, and L. Mitchell. Data Assimilation Using an Ensemble Kalman Filter Technique, 1998.

[13] W. keng Liao, A. Ching, and K. C. etc. An Implementation and Evaluation of Client-Side File Caching for MPI-IO. In *IPDPS*, 2007.

[14] W.-k. Liao and A. Choudhary. Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols. SC'08, 2008.

[15] H. Lim, V. Kapoor, C. Wighe, and D. Du, H.-C. Active Disk File System: a Distributed, Scalable File System. In *18th IEEE Symposium on Mass Storage Systems and Technologies (MSS '01.)*, pages 101–116, San Diego, CA, USA, 2001. IEEE Computer Society.

[16] J. F. Lofstead, S. Klasky, and K. e. Schwan. Flexible IO and Integration for Scientific Codes through the Adaptable IO System (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008.

[17] X. Ma, M. Winslett, J. Lee, and S. Yu. Faster Collective Output through Active Buffering. In *IPDPS*, 2002.

[18] B. Nitzberg and V. Lo. Collective Buffering: Improving Parallel I/O Performance. In *High-Performance Distributed Computing, International Symposium on*, 1997.

[19] NOAA. Overview of Current Atmospheric Reanalysis. http://reanalyses.org/atmosphere/overview-current-reanalyses, 2012.

[20] R. Oldfield, L. Ward, R. Riesen, A. B. Maccabe, P. Widener, and T. Kordenbrock. Lightweight i/o for scientific applications. In *CLUSTER*, 2006.

[21] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *SOSP*, 1995.

[22] L. Qin and D. Feng. Active Storage Framework for Object-based Storage Device. In *20th International Conference on Advanced Information Networking and Applications*, pages 97–101, 2006.

[23] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *In Proceedings of the 24rd International Conference on Very Large Data Bases(VLDB '98)*, 1998.

[24] M. Sivathanu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Evolving RPC for Active Storage. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002.

[25] C. W. Smullen, S. Rohinton, S. Gurumurthi, P. Ranganathan, and M. Uysal. Active Storage Revisited : The Case for Power and Performance Benefits for Unstructured Data Processing Applications. *Proceedings of the 5th conference on Computing frontiers*, 2008.

[26] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective i/o in romio. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, 1999.

[27] N. Tran and D. A. Reed. Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching. *IEEE Transactions on Parallel and Distribted Systems*, 15:362–377, 2004.

[28] M. Vilayannur, A. Sivasubramaniam, M. Kandemir, R. Thakur, and R. Ross. Discretionary Caching for I/O on Clusters. *Cluster Computing*, 9:29–44, 2006.

[29] S. Woo, S. Samuel, L. Philip, C. Robert, and R. Rajeev. Enabling Active Storage on Parallel I/O Software Stacks. In *26th IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[30] Y. Xie, D. Feng, and D. D. E. Long. Design and Evaluation of Oasis : An Active Storage Framework based on TIO OSD Standard. In *27th IEEE Symposium on Mass Storage Systems and Technologies*, 2011.

[31] W. Yu and J. Vetter. Parcoll: Partitioned collective i/o on the cray xt. In *ICPP*, 2008.

[32] X. Zhang, S. Jiang, and K. Davis. Making resonance a common case: A high-performance implementation of collective i/o on parallel file systems. In *IPDPS*, 2009.

[33] F. Zheng, H. Abbasi, and C. D. etc. Predata - preparatory data analytics on peta-scale machines. In *IPDPS*, pages 1–12, 2010.