Speed Up Large Scale Data Visualization via a Distributed Storage Infrastructure

Willie Ng, Yongqing Zhu, Yang Yu, Samsudin Juniarto

*Data Storage Institute, A*STAR (Agency for Science, Technology and Research), Singapore,*
Tel: (65) 6408 2988, Fax: (65) 6516 0900
Email: Willie_NG@dsi.a-star.edu.sg, ZHU_Yongqing@dsi.a-star.edu.sg, YU_Yang@dsi.a-star.edu.sg,
Juniarto_SAMSUDIN@dsi.a-star.edu.sg

***Abstract*— The push to introduce faster supercomputers is generating ever-larger data sets, challenging traditional methods of data analysis and visualization. While the computational power of supercomputers keep increasing with every generation, the I/O systems have not kept pace, resulting in a significant performance bottle neck. This has greatly hampered the performance of data analysis and visualization in big data era.**

We propose a solution, VisDSI (Visualization via a Distributed Storage Infrastructure), to address the problem and eliminate I/O bottlenecks by 1) using traditional high performance clusters with disks directly attached within each node; 2) deploying a data-intensive distributed file system on the cluster and exploiting data locality information to visualization; and 3) developing a POSIX-compatible I/O layer to enable the traditional visualization applications to smoothly port to this new platform as well as to provide a new I/O semantic of retrieving the data location with respect to the POSIX standards. In particular, VisDSI guarantees the co-located compute and data storage by introducing a scheduling of work assignments to nodes with local copies of needed data. Compared with the original visualization application, our solution runs at least 10 times faster.

***Keywords*— I/O Scheduling; Distributed File System; Parallel and Distributed Computing; Data Visualization**

I. INTRODUCTION

The computational science community is approaching petascale level simulations that will produce extremely large datasets. Processing these datasets are both critical and challenging as we require substantial hardware resources for simulation, data storage, and data analysis. While the computational power of supercomputers keep increasing, the I/O systems have not kept pace, resulting in a significant performance bottle neck. Childs, et. al. [1] demonstrated visualization performance at scale on six supercomputers across the United States Department of Energy complex. An important finding made was that although the actual visualization algorithm and rendering times were sufficiently fast for interactive performance even with ultra-scale datasets, I/O time dwarfed these operations by at least an order of magnitude. In another observation by Ross, et. al. [2], the I/O could consume nearly 90% of the total time of ParaView [3] volume-rendering for a 16002-pixel image in a multi-core clusters served by a parallel storage system.

In this paper, we opt in favor of moving processing closer to data, a technique suggested by Mitchell, et. al. [4]. The challenge of this approach is the porting of visualization applications from the traditional computational-intensive platforms to the data-intensive platforms. To keep pace with the scale of data, we propose a solution, VisDSI (Visualization via a Distributed Storage Infrastructure). We replaced the centralized parallel storage system with a distributed file system with local hard drives directly attached to individual nodes of the cluster [5]. VisDSI addresses the problem by

1) using traditional high performance clusters with disks directly attached within each node;

2) deploying a data intensive file system on the cluster and exploiting data locality information to visualization;

3) developing a POSIX-compatible I/O layer to enable the traditional visualization applications to smoothly port to this new platform as well as to provide a new I/O-semantic of retrieving the data location with respect to the POSIX standards.

VisDSI guarantees the co-located compute and data storage by introducing a data-locality aware algorithm. Our algorithm assigns task to nodes with local copies of needed data. With the algorithm, our solution allows MPI-based applications, such as ParaView, to interpret the data locality information exposed by the file system

and appropriately schedule computation on nodes locally containing the needed data.

II. VISDSI SOLUTION

To enable visualization applications to run on data intensive clusters and leveraging in the data locality, there are three issues we need to consider: firstly, we need to make the visualization application talk to the underlying data-intensive file systems through POSIX interfaces. This is to facilitate both accessing data and retrieving data location; secondly, we need to improve the original static scheduling mechanism in visualization applications into a dynamic scheduling mechanism; and thirdly, we need propose a new locality-aware scheduling policy that considers both data locality and load balancing.

A. POSIX-IO Support

In this paper, we employed ParaView, an open-source, multi-platform visualization application. ParaView is a typical MPI-based HPC application, which works on HPC cluster and stores files into parallel file system through normal POSIX-IO. POSIX (Portable Operating System Interface) is a family of standards specified by IEEE for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software to be compatible with variants of Unix and other operating systems. POSIX-IO is the I/O parts of the POSIX standards, which includes both the specific interfaces like open/read/write/Iseek/close and their semantics. Distributed file system like HDFS adopts a client-server model, in which the servers manage the data/metadata while the clients request data from servers. HDFS is not fully POSIX compliant, because the requirements for a POSIX file system differ from the target goals for a Hadoop application [5].

In order to support POSIX-IO, the client of a distributed file system has to be wrapped as a local file system. Currently, there are two ways to achieve this goal. One way is to wrap the client as a standard kernel-level file system. The other way is to wrap the client as a fuse user-level file system. The former is directly implemented in kernel with the advantage of good performance, but it increases the difficulty of programming and debugging. The latter is implemented in the user-level, which is simpler than kernel but may have small performance penalty. In order to achieve POSIX-IO support without too much implementation difficulty, we adopt the latter way to wrap a HDFS client at use level (see Figure 1).

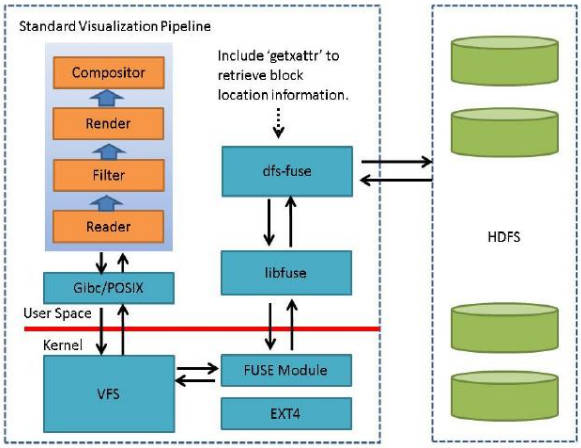

Figure 1: Support for POSIX-IO

Fuse is a user-level file system framework, which simplifies the development of a new local file system. It has three components: kernel-module, utility tools, and a libfuse library. Developers can implement a file system in the userlevel with the help of libfuse library. The libfuse library can communicate with utility tools to mount the user-level file system. After it is mounted, all the file system requests to the mounted directory would be directed to the kernelmodule, which would then direct these requests to the libfuse library through /dev/fuse. The libfuse library would direct the requests to the user-level file system. The responses of user-level file system would be sent back through the reverse direction.

B. Locality-aware Scheduling

Currently, visualization applications like ParaView use a static scheduling mechanism to assign computation tasks to processes. The dataset is partitioned equally into, *n* pieces, where *n* is the number of processes, then each process would be assigned a piece of data according to their MPI rank. This static scheduling mechanism is straightforward and works well on computation-intensive clusters with the support of a parallel file system. However, when we move the visualization to a data-intensive cluster, the static scheduling mechanism cannot leverage the data locality feature of HDFS. In order to address the problem, we propose a locality-aware dynamic scheduling algorithm (see Figure 2).

The locality-aware scheduling algorithm is used to move computation to the DataNodes with the required data as well as to maintain load balancing to improve

the overall system performance. The main objective is to achieve highest system overall performance by both considering data locality and load balancing. The conflict between data locality and load balancing is that the data distribution is not always even because of HDFS's random chunk allocation policy. So if we only schedule computation to the node with the required data, some node may get heavy loads while others may get light loads. This imbalance will cause the low efficiency of computation resources, and eventually will hurt the overall system performance. By defining the relationship between computation-overhead and data-transfer overhead, we can create a theory to balance both and achieve the best system overall performance.

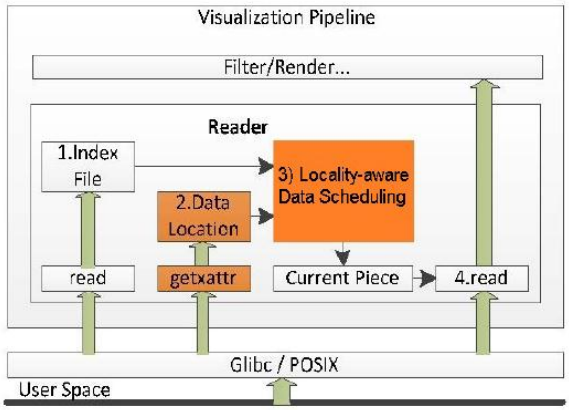

Figure 2: Dynamic Scheduling Mechanism

Suppose a visualization pipeline will render a file, then the total time it will cost is:

$$t_{total} = max\{t_{node1}, t_{node2}, \ldots, t_{noden}\} + t_{sub\text{–}image\ transfer} + t_{composite} \quad (1)$$

For a single DataNode, the time of rendering a specified file is:

$$t_{node} = t_{read} + t_{filter} + t_{render} \quad (2)$$

If the file is read from local disk:

$$t_{localread} = t_{diskprepare} + s_{file}/B_{disk} \quad (3)$$

If the file is read from network

$$t_{remoteread} = t_{localread} + t_{networkprepare} + s_{file}/B_{network} \quad (4)$$

For a specified dataset, the $t_{sub\text{–}image\ transfer}$ and $t_{composite}$ can be seen as fixed. Since the visualization is a data-intensive, t_{node} is mainly decided by t_{read}. So in order to make the entire visualizing time smallest, we need to make the t_{node} as low as possible. We consider $t_{remoteread} \gg t_{localread}$ as a remote read need to include the network transfer time. According to this, we propose our locality-aware scheduling algorithm as shown in Algorithm 1. First, it gives priority to the DataNodes based on a data locality policy. For a given computation task, this policy gives higher priority to any node with local copy of needed file. Second, it assigns the task to one of these nodes based on a minimal load policy. The second policy helps in minimizing the chances of assigning a task to a node that is heavily loaded. The minimal load balance is a little more complex. For a given file, it will calculate the new load of each node assuming these nodes have been assigned the file, and then select the node with the smallest load. This process will run in loop until all the files have been assigned.

Algorithm 1: Locality-aware Data Scheduling

Input :
- N: a set of nodes available;
- F: a set of files making up a time step;
- f: an element of F;
- L: a set of tuples, $< file, node >$, denoting the physical location of f and its replicas;
- $s(f)$: represents the size of the f;
- B_{disk}: disk bandwidth;
- $B_{network}$: network bandwidth;
- $t_{disk_prepare}$: waiting time for disk;
- $t_{network_prepare}$: waiting time for network;

Output:
- R: a set of tuples, $\langle node, file \rangle$, denoting which files are assigned to a specified node;

```
1  begin
2      R ← ∅;
3      C_load(N) is a function which reflects the node to its current
       load, initialized with 0;
4      while F ≠ ∅ do
5          find L for f;   // the location of f and its replicas
6          C_read(n) is a function which reflects the time cost of a
           specified node to read a specified file, initialized with 0;
7          N' ← N;
8          while N' ≠ ∅ do
9              n ∈ N'
10             if ⟨f, n⟩ ∈ L then
11                 C_read(n, f) ← t_diskprepare + s(f)/B_disk;
12             else
13                 C_read(n, f) ← t_diskprepare + s(f)/B_disk +
                   t_networkprepare + s(f)/B_network;
14             end
15         end
16         find n ∈ N with the smallest value of
           C_load(n) + C_read(n, f);
17         R ← R ⋃ {(n, f)};
18         F ← F − {f};
19         C_load(n) ← C_load(n) + C_read(n, f);
20     end
21 end
```

III. EVALUATION

We have set up a Hadoop cluster to evaluate the performance of our VisDSI solution. The cluster was built using 19 nodes (heterogeneous). All the machines are commodity hardwares. The network bandwidth is 1Gb. In order to support POSIX-IO, we installed fuse dfs on top of the HDFS. After mounting the HDFS, we installed ParaView in every node. VisDSI is implemented in our locality-aware scheduling library and compiled as a dynamic link library. Therefore, we can change the scheduling algorithm by simply update

the dynamic library. To evaluate the performance of VisDSI, we compared the system having deployed the locality-aware scheduling algorithm with the original ParaView that runs on top of HDFS. Testing was conducted using ParaView 3.14.1 and Hadoop version 1.0.3. Additionally, Hadoop's configuration was kept as close to default as possible (for example, 3 replica per file and 64MB per block).

The XML Multi-Block Data reader reads the VTK XML multi-block data file format. XML multi-block data files are meta-files that point to a list of serial VTK XML image data files (.vti). We wrote a program that can easily generate multiple XML data files. Here, each file represents a rectangular box having unique dimensions (see Figure 3). Since we are studying the I/O performance, the XML file needs to be large. We achieved this by adding multiple lines of comment onto the data files. In this way, each file is able to fill up a block (64 MB) in HDFS. In total, we created about 17,000 image files and they were pumped into the HDFS. This was equivalent to 1TB.

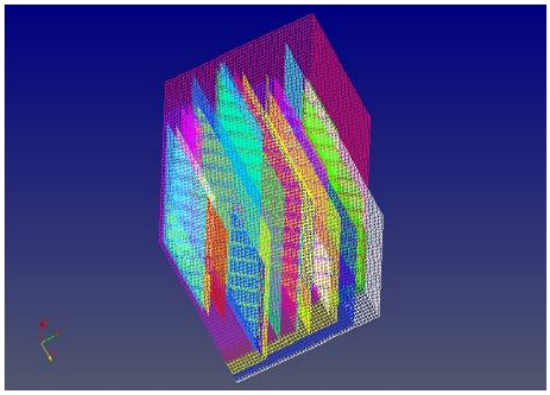

Figure 3: 3D image generated by 100 image files

For the test, we will denote VisDSI as system having the locality-aware scheduling algorithm and denote Original PV as the original ParaView. To see if VisDSI is helping the reader to read and process data locally, we let the two systems run on five test points. For each test point, we monitor the data movement across the network. If a node reads a file from another node, the file that is transferred will be recorded. The result is shown in Figure 4. It becomes clear that VisDSI is able to achieve 100% file locality. There is no transfer of file from node to node. On the other hand, without the locality-aware algorithm in operation, only 20% of the required data is co-located with the task nodes. Original PV needs to fetch the rest of the files remotely across the network. Consider the last test point containing 17,000 files (1 TB). Each compute node on average will need to pull at least 750 files from the network. This will inevitably create high congestion in the network.

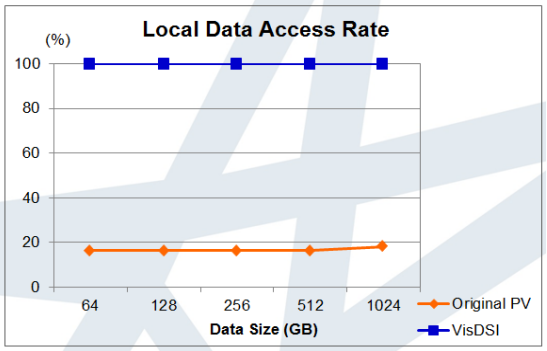

Figure 4: Local data access rate of the Original ParaView vs. VisDSI

Figure 5 shows the visualization performance of both Original PV and VisDSI.. Note that we used logarithmic scaling on the vertical axis. The response time of the two systems were computed as an average of 5 runs with each run having 5 test points. The response time refers to the time spent starting from the instant the client made the request until the image was displayed. Previously, we have seen that the DataNodes in Original PV requires to load files remotely. Since $t_{remoteread} \gg t_{localread}$, the overhead incurred when transferring large data across the network will be significant. Logically, if the number of remote accessed file is large, we will see a surge in the overall system time and this is clearly demonstrated in Figure 5. Without the locality-aware scheduling algorithm, the system can be slower by an order of magnitude. With 1TB of dataset, Original PV took more than 6 hours to complete the task while VisDSI took only 30 minutes. From this experiment, data locality indeed plays a very important role in improving the I/O Bandwidth.

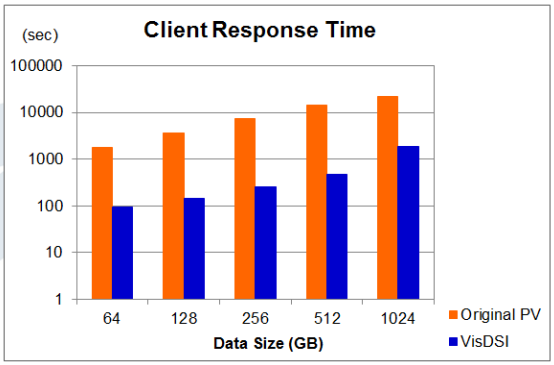

Figure 5: Visualization Performance of the Original ParaView vs. VisDSI

V. RELATED WORKS

There have been many strategies developed to address the I/O bottlenecks in computer science. The basic idea of collective I/O is to merge small, non-contiguous disk accesses into large, contiguous disk accesses by multiple processors. After each processor fetches a chunk of the data from disk, through the interconnection network data items are routed to the processor requests them. Such two-phase I/O approaches were first introduced by Rosario et al. [6], and then extended by Thakur et al. in ROMIO [7]. Kotz [8] introduced disk-directed I/O which moves the I/O management job from the client program running on the compute processors to the disk server. Disk-directed I/O requires no communication among I/O processors nor among compute processors.

The objective of data sieving is similar to that of collective I/O. With data sieving, the fetching of several non-contiguous pieces of data is replaced by a single read of a large contiguous chunk of the data containing all those individual pieces. In most cases, the benefit of reading large, contiguous chunks of data far outweighs the cost of reading unwanted data [9], but the memory overhead to store the unneeded data for data sieving can become excessive. ROMIO [7] provides a user-controllable parameter to define the maximum amount of contiguous data that a process can read at a time with data sieving.

MPI-IO [10] provides both portability and convenience to perform parallel file I/O. It defines a set of routines for transferring data to and from external storage devices, and supports a number of useful parallel I/O features including collective I/O. Peterka, et. al. [11] founds that the NetCDF [12] format can perform poorly with the two phase optimization of MPI-IO, while using HDF51291 or netCDF-64bit can improve.

Ross [2] described a visibility-aware data distribution strategy for parallel file system to eliminate the non-contiguous I/O access patterns on the underlying I/O servers. The strategy is to group all the data with the same visibility patterns together into a file, and then stripe the file along a sequence of I/O servers in a round-robin manner. The strategy can achieve both maximum I/O server parallelism and contiguous I/O within individual I/O servers. The visibility patterns are calculated through visibility feature vectors, which are constructed based on the data itself.

Mitchell proposed VisIO [4] to provide linear scalability of I/O bandwidth for ultra-scale visualization by leveraging the data locality. The solution develops a VisIO library for visualization application to access the dataset in HDFS. Our VisDSI solution is different from [4] in that we develop the POSIX-compatible I/O layer on top of FUSE as well as the locality-aware scheduling for visualization application.

IV. CONCLUSION

The performance mismatch between computing and I/O components of current generation HPC systems has caused I/O the critical bottleneck for many scientific visualization applications. In this paper, we have proposed a VisDSI solution to address this issue. VisDSI replaces the centralized parallel storage system with a distributed file system having local hard drives directly attached to individual nodes of the cluster. It allows traditionally POSIX and MPI based visualization applications to leverage the increased bandwidth possible from the distributed file system. By introducing a locality-aware scheduling of work assignments to nodes with local copies of needed data, we efficiently reduce the read time of data.

Our solution works as a middle layer. Very little code is modified based on both file system level and application level. VisDSI does not require sophisticated machines. Testing was conducted using commodity hardware and our solution was proven to be able to improve the I/O bandwidth for large scale data visualization. Compare to the original ParaView, our solution runs at least 10 times faster.

REFERENCES

[1] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. H. Weber, and E. W. Bethel, "Extreme scaling of production visualization software on diverse architectures," *IEEE Computer Graphics and Applications*, vol. 30, no. 3, pp.22–31, 2010.

[2] R. B. Ross, T. Peterka, H.-W. Shen, Y. Hong, K.-L. Ma, H. Yu, and K. Moreland, "Visualization and parallel I/O at extreme scale," *Journal of Physics: Conference Series*, vol. 125, no. 1, p. 012099, 2008.

[3] "Paraview," 2013. [Online]. Available: http://www.paraview.org/

[4] C. Mitchell, J. P. Ahrens, and J. Wang, "VisIO: Enabling interactive visualization of ultra-scale, time series data via high-bandwidth distributed i/o systems," in *IPDPS*, 2011, pp.68–79.

[5] "Apache Hadoop," 2013. [Online]. Available: http://wiki.apache.org/hadoop/

[6] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a two-phase run-time access

strategy," SIGARCH Comput. Archit. News, vol. 21, no. 5, pp. 31–38, Dec. 1993.

[7] 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, ser. FRONTIERS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–191.

[8] D. Kotz, "Disk-directed I/O for mimd multiprocessors," in Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation, ser. OSDI '94. Berkeley, CA, USA: USENIX Association, 1994.

[9] R. Thakur, W. Gropp, and E. Lusk, "A case for using MPI's derived datatypes to improve I/O performance," in Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM), ser. Supercomputing '98, 1998, pp. 1–10.

[10] W. Gropp, R. Thakur, and E. Lusk, Using MPI-2: Advanced Features of the Message Passing Interface, 2nd ed. Cambridge, MA, USA: MIT Press, 1999.

[11] T. Peterka, H. Yu, R. Ross, K.-L. Ma, and R. Latham, "End-to-End study of parallel volume rendering on the IBM Blue Gene/P," in Proceedings of the 2009 International Conference on Parallel Processing, ser. ICPP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 566–573.

[12] J. Li, W.-k. 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 Proceedings of the 2003 ACM/IEEE conference on Supercomputing, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 39