Efficient Disk-to-Disk Sorting: A Case Study in the Decoupled Execution Paradigm

Hassan Eslami[1], Anthony Kougkas[2], Maria Kotsifakou[1], Theodoros Kasampalis[1]
Kun Feng[2], Yin Lu[3], William Gropp[1], Xian-He Sun[2], Yong Chen[3], Rajeev Thakur[4]
[1]University of Illinois at Urbana-Champaign, [2]Illinois Institute of Technology
[3]Texas Tech University, [4]Argonne National Laboratory
{eslami2, kotsifa2, kasampa2, wgropp}@illinois.edu
{akougkas, kfeng1}@hawk.iit.edu, sun@iit.edu
{yin.lu, yong.chen}@ttu.edu, thakur@mcs.anl.gov

ABSTRACT

Many applications foreseen for exascale era should process huge amount of data. However, the IO infrastructure of current supercomputing architecture cannot be generalized to deal with this amount of data due to the need for excessive data movement from storage layers to compute nodes leading to limited scalability. There has been extensive studies addressing this challenge. Decoupled Execution Paradigm (DEP) is an attractive solution due to its unique features such as available fast storage devices close to computational units and available programmable units close to file system.

In this paper we study the effectiveness of DEP for a well-known data-intensive kernel, disk-to-disk (aka out-of-core) sorting. We propose an optimized algorithm that uses almost all features of DEP pushing the performance of sorting in HPC even further compared to other existing solutions. Advantages in our algorithm are gained by exploiting programming units close to parallel file system to achieve higher IO throughput, compressing data before sending it over network or to disk, storing intermediate results of computation close to compute nodes, and fully overlapping IO with computation. We also provide an analytical model for our proposed algorithm. Our algorithm achieves 30% better performance compared to the theoretically optimal sorting algorithm running on the same testbed but not designed to exploit the DEP architecture.

Keywords

Decoupled execution paradigm, disk-to-disk sorting, performance optimization, parallel file system, parallel IO

1. INTRODUCTION

Many scientific applications running on HPC systems are dealing with ever increasing amount of data [10, 24]. In many of these applications most of the execution time is spent in IO where large amounts of data is written to and/or read from storage layers [22]. HPC systems offer a fast interconnection network and flexible IO infrastructure that makes them very attractive for many Big Data analytics as well [6, 12, 16–18, 20, 27]. Large-scale machine learning and graph analytics are just a few examples of such analytics that has arisen in current HPC systems [2, 11].

On the other hand, current HPC systems are not well-suited for emerging data-intensive applications. Not only is the computation rate of current multi-core/many-core architectures greater than the data access rate in storage devices, but also the improvements in computation rate follow a faster trend compared to the improvements in data access rate [13, 15]. This causes the so called "IO-wall" problem in which the gap between computation rate and data access rate grows continuously.

Extensive research is done to propose architectural improvements to the current HPC environments for data-intensive applications. One proposed solution is the use of more suitable devices such as solid-state drives (SSD) and phase-change memories (PCM) in the IO hardware stack [8, 21]. Although this solution reduces the gap between computation rate and data access rate, it still does not solve the IO-wall problem. Another proposed solution is the addition of computational units closer to the actual data [5, 26, 29]. This solution enables decoupling and shipping data-intensive operations closer to data, reducing data movement all the way across the network from storage layers to main computational units. For most users, however, available computational units closer to data usually have limited capability and lack flexibility in programming, making the second solution difficult to exploit.

One promising system architecture proposed for data-intensive computing is the Decoupled Execution Paradigm (DEP) [7, 9]. DEP decouples nodes into Data Nodes (DN) and Compute Nodes (CN). CNs are attached via a fast network and do not have any persistent storage device. These nodes are similar to main computational units in commonly used current supercomputing architectures, and are suitable for computational intensive operations. DNs are usually beefy nodes with local SSDs suitable for data-intensive operations. Data nodes are decoupled further into Compute-side Data Nodes (CDN) and Storage-side Data Nodes (SDN). SDNs are connected to storage devices with a fast network, and similarly, CDNs are connected to CNs with a fast network. With these programmable beefy DNs, DEP offers

DISCS-2015, November 15-20, 2015, Austin, TX, USA
© 2015 ACM 978-1-4503-3993-3/15/11 ...$15.00.
DOI: http://dx.doi.org/10.1145/2831244.2831249

generality and flexibility in data-intensive operations that other data-intensive architectures do not provide in an easy way.

In this paper we investigate the effectiveness of DEP, both analytically and practically, through a well-known data-intensive kernel, **disk-to-disk sorting**. Our algorithm is designed for efficient sorting of a data-set that is too large to fit into aggregate available RAM[1]. Our contributions are the following:

- We introduce a new sorting technique in the context of DEP that effectively uses almost all features of the DEP system architecture. We demonstrate that the total execution time of our algorithm is almost equivalent to the aggregate time needed to read the entire data from disk and write it back to disk. In other words, we show that the actual computation (sorting) and reading/writing of intermediate data is almost entirely overlapped with reading the input and writing the output.
- We analytically model the proposed sorting algorithm to justify our design choices, and show the accuracy of our model.
- We demonstrate that our approach in DEP achieves 30% better performance compared to the *theoretically optimal* sorting algorithm running on the same testbed but not exploiting the DEP architecture.

2. DISK-TO-DISK SORTING ALGORITHM

Our design of disk-to disk sorting was driven by the need to utilize all the different parts of the DEP architecture. SDNs are connected to the file system, therefore will be responsible for reading the input, distributing it appropriately to the remaining nodes in the system, and writing the final result back to the file system. CDNs, being connected to fast local SSDs, will be responsible for storing intermediate results. Finally, the CNs, being the most numerous type of nodes in the DEP architecture, will perform the main part of the computation.

The disk-to-disk sorting algorithm is divided into two phases, the read phase and the write phase. During the read phase, input data is read by the SDNs and distributed to the CNs, which perform some initial processing and send the data for intermediate storage to the CDNs. The end of the input data identifies the end of the read phase and the beginning of the write phase. During the write phase, CDNs send the intermediate results to CNs in an appropriate order. The CNs then compute the final result and send the sorted data to SDNs for storage.

The algorithm is capable of handling skewed data distribution using similar sampling technique discussed in [28]. Data sampling can be done amongst SDNs right before the first time they communicate with CNs. However, for the sake of simplicity, the algorithm we describe here (and also the analysis in Section 3) assumes that the input data follows a uniform distribution with known minimum and maximum values. Hence, we assume that throughout the entire sorting process, each CN deals only with a specific predefined subrange of the data. In order to achieve balance in the amount of data being processed by each CN, the subrange of the i-th CN is defined as $[\frac{i(max-min)}{NUM\_CN}, \frac{(i+1)(max-min)}{NUM\_CN} - 1]$. Also, since the CNs are the most numerous type of nodes, each SDN will eventually collect sorted data from a set of CNs with consequtive assigned subranges. The set of CNs assigned to each SDN is predefined, based on an internal ranking of the SDNs.

The following sections provide a more detailed description of operations done in different types of nodes during the read and write phases of the algorithm.

2.1 Read Phase

During the read phase (algorithms 1, 2, and 3), the SDNs read data from the disk in batches, sort these batches, and send them to the appropriate CNs after compressing them. The SDNs use non-blocking MPI-IO read calls and double-buffering to efficiently read data through the storage device network. Each SDN overlaps reading the next data batch with processing of the current data batch. The processing involves sorting, partitioning, and compressing the data. First, the current data batch is sorted into sorted runs using local thread parallelism. The sorted runs in turn are merged with the already sorted data found in the SDN's memory (represented by the `Mem1` and `Mem2` buffers in the pseudocode). When the SDN's memory is full, the sorted data are partitioned according to the CN subranges and each partition is compressed and sent to the corresponing CN. Again, the sending time is overlapped with the combined IO, processing, and compressing time. Moreover, sending compressed sorted data allows for smaller messages as the compression ratio for sorted data is higher.

Each CN waits for messages of sorted data sent from SDNs. While waiting for the next message, it decompresses and merges the latest received data with data already found in its memory. When the CN's memory is full, it sends the data to its assigned CDN as a series of compressed chunks. The size of the chunk is a user defined constant and the function call $chunk(i)$ computes the range of the i-th chunk. Note that each series of chunks contains sorted data belonging to the CN's predefined range. The compressing plus sending opeartions are overlapped with the rest of the operations.

Finally, each CDN collects data from all its assigned CNs and stores them in a file on its local SSD. The file is indexed by CN rank, series number, and chunk number for easy extraction of the information. The read phase ends when all input data has been read by the SDNs, processed through the CNs, and stored in the form of sorted series of chunks in the SSDs of the CDNs.

2.2 Write Phase

During the write phase (algorithms 4, 5, and 6), each CDN serves requests from its assigned CNs for chunks from specific series. The requests come in the form of series numbers. The CDN keeps track of the last sent chunk of each series for each CN and always sends the next chunk in order.

Each CN merges data from all the series it produced during the write phase into a merge buffer (represented by the `MergeBuf` buffer in the pseudocode). Initially, the CN requests the first chunk of each series and then it performs a multi-way merge of all these chunks (lines 11-36 of algorithm 5). If all the elements of a chunk have been merged into the result, the next chunk of these series is requested (lines 15-29 of algorithm 5). When the merge buffer is full,

[1]Code and full report at: github/heslami/dep-sort

Algorithm 1 Storage-side Data Node Read Phase

```
1:                                                          ▷ Initialization
2: ReadBuf ← READ()
3:                                                          ▷ Main Loop
4: while not EOF found do
5:     SWAP(ReadBuf, DataBuf)
6:     ReadBuf ← IREAD(ReadReq)
7:     SortedRunsBuf ← PARALLELSORT(DataBuf)
8:     SortedBuf ←
9:         MULTIWAYMERGE(SortedRunsBuf, SortedBuf)
10:    if SortedBuf.lenght + Mem1.length > Mem2.size then
11:        WAITALL(SendReqs)
12:        for each Compute Node CN, in parallel do
13:            SendBufs[CN] ← COMPRESS(Mem1[range(CN)])
14:            ISEND(SendBufs[CN], CN, SendReqs[CN])
15:        Mem1.clear()
16:    Mem2 ← MERGE(SortedBuf, Mem1)
17:    SWAP(Mem1, Mem2)
18:    WAIT(ReadReq)
```

Algorithm 2 Compute Node Read Phase

```
1:                                                          ▷ Initialization
2: NumSeries ← 1
3: RecvBuf ← RECV(FROM_ANY_SDN)
4:                                                          ▷ Main Loop
5: while not received finisher do
6:     SWAP(RecvBuf, CompDataBuf)
7:     RecvBuf ← IRECV(FROM_ANY_SDN, RecvReq)
8:     DataBuf ← DECOMPRESS(CompDataBuf)
9:     if DataBuf.length + Mem1.length > Mem2.size then
10:        WAITALL(SendReqs)
11:        NumChunks ← Mem1.length / CHUNK_SIZE
12:        for i = 1 to NumChunks, in parallel do
13:            SendBufs[i] ← COMPRESS(Mem1[chunk(i)])
14:        for i = 1 to NumChunks do
15:            msg ← {NumSeries, i, SendBuffs[i]}
16:            ISEND(msg, TO_MY_CDN, SendReqs[i])
17:        Mem1.clear()
18:        NumSeries ← NumSeries + 1
19:    Mem2 ← MERGE(DataBuf, Mem1)
20:    SWAP(Mem1, Mem2)
21:    WAIT(RecvReq)
```

Algorithm 3 Compute-side Data Node Read Phase

```
1:                                                          ▷ Initialization
2: finished ← 0
3: for each compute node CN assigned to this CDN do
4:     MsgBufs[CN] ← IRECV(from CN, RecvReqs[CN])
5:                                                          ▷ Main Loop
6: while finished < assigned do
7:     CN ← WAITANY(RecvReqs)
8:     if received finisher then
9:         finished ← finished + 1
10:        continue
11:    series ← MsgBufs[CN].series
12:    chunkNo ← MsgBufs[CN].chunkNo
13:    SWAP(MsgBufs[CN].chunkBuf, ToFileBuf)
14:    MsgBufs[CN] ← IRECV(CN, RecvReqs[CN])
15:    WRITETOLOCALFILE(CN, series, chunkNo, ToFileBuf)
```

Algorithm 4 Storage-side Data Node Write Phase

```
1:                                                          ▷ Initialization
2: finished ← 0
3: RecvBuf ← IRECV(FROM_ANY_CN)
4:                                                          ▷ Main Loop
5: while finished < assigned do
6:     CN ← RecvBuf.source
7:     SWAP(RecvBuf, CompDataBuf)
8:     RecvBuf ← IRECV(FROM_ANY_CN, RecvReq)
9:     if received finisher then
10:        finished ← finished + 1
11:        continue
12:    WAIT(WriteReq)
13:    DataBuf ← DECOMPRESS(CompDataBuf)
14:    offs ← CALCULATEOFFSET(CN)
15:    IWRITE_AT(DataBuf, offs, WriteReq)
16:    WAIT(RecvReq)
```

Algorithm 5 Compute Node Write Phase

```
1:                                                          ▷ Initialization
2: NumSeriesFinished ← 0
3: for s = 1 to NumSeries do
4:     SeriesIndices[s] ← 0
5:     SeriesFinished[s] ← false
6:     SEND(s, TO_MY_CDN)
7:     RecvBufs[s] ← IRECV(FROM_MY_CDN, RecvReqs[s])
8:                                                          ▷ Main Loop
9: while NumSeriesFinished < NumSeries do
10:    for i = 1 to MergeBuf.size do
11:        min ← inf
12:        for s = 1 to NumSeries do
13:            if SeriesFinished[s] then continue
14:            si ← SeriesIndices[s]
15:            if si > SeriesBufs[s].length then
16:                WAIT(RecvReqs[s])
17:                if received series finisher then
18:                    NumSeriesFinished ←
19:                     NumSeriesFinished + 1
20:                    SeriesFinished[s] ← true
21:                    continue
22:                SWAP(RecvBufs[s], CompSeriesBufs[s])
23:                SEND(s, TO_MY_CDN)
24:                RecvBufs[s] ←
25:                    IRECV(FROM_MY_CDN, RecvReqs[s])
26:                SeriesBufs[s] ←
27:                    DECOMPRESS(CompSeriesBufs[s])
28:                si ← 1
29:                SeriesIndices[s] ← 1
30:            if SeriesBufs[s][si] < min then
31:                min ← SeriesBufs[s][si]
32:                minseries ← s
33:        if NumSeriesFinished = NumSeries then break
34:        MergeBuf[i] ← min
35:        SeriesInidices[minseries] ←
36:            SeriesInidices[minseries] + 1
37:    if MergeBuf.length = 0 then break
38:    WAIT(SendReq)
39:    SWAP(MergeBuf, SendBuf)
40:    CompSendBuf ← COMPRESS(SendBuf)
41:    ISEND(CompSendBuf, TO_MY_SDN, SendReq)
42:    MergeBuf.clear()
```

Algorithm 6 Compute-side Data Node Write Phase

```
1:                                                    ▷ Initialization
2: finished ← 0
3: for each compute node CN assigned to this CDN do
4:     ReqSeries[CN] ← IRECV(CN, RecvReqs[CN])
5:     RemainingSeries[CN] ← FileIndex[CN].length
6:     for s = 1 to RemainingSeries[CN] do
7:         NextChunk[CN][s] ← 0
8:                                                    ▷ Main Loop
9: while finished < assigned do
10:    WAITANY(RecvReqs)
11:    CN ← RecvReqs.source
12:    series ← ReqSeries[CN]
13:    chunkNo ← NextChunk[CN][s]
14:    if chunkNo > FileIndex[CN][series].length then
15:        SENDSERIESFINISHER(CN)
16:        RemaingSeries[CN] ←
17:            RemaingSeries[CN] - 1
18:        if RemaingSeries[CN] = 0 then
19:            finished ← finished + 1
20:        else
21:            ReqSeries[CN] ← IRECV(CN, RecvReqs[CN])
22:        continue
23:    WAIT(SendReq)
24:    SendBuf ← READFROMLOCALFILE(CN, series, chunkNo)
25:    ISEND(SendBuff, CN, SendReq)
26:    NextChunk[CN][series] ← chunkNo + 1
27:    ReqSeries[CN] ← IRECV(CN, RecvReqs[CN])
```

it is compressed and sent to the appropriate SDN. The time spent in compressing and sending is overlapped with the rest of the operations. Also, the time for receiving a new chunk is overlapped with the process of merging. The stream of merge buffers being sent contains all the input data belonging to the CN's subrange in sorted order.

Each SDN receives sorted data buffers from its assigned CNs, decompresses it and stores them in the disk using non-blocking MPI-IO writes. The SDNs are responisble for storing data received from different CNs in the correct order according to CNs' subranges. The combined decompressing and writing time is overlapped with the time between subsequent receives from CNs.

3. PERFORMANCE MODELING

In this section we devise a simple performance model for the read phase of SDNs. This is only one component out of the six components of our algorithm. Due to lack of space, we ignore the performance model of other five components, but those can be derived in a similar way. Derived performance model for all the components can be found in the source code repository of the project. Our analysis is based on system and algorithm specific parameters shown in Table 1 and Table 2. Note that we assume zero network latency in our modeling as almost all messages in our algorithm are large.

For **SDNs**, there are four major operations:

- Reading a buffer of size SDN_READ_SIZE from the file system.
- Sorting the read buffer and merging it with the in-memory buffer (with maximum size of SDN_BUFFER_SIZE). We used quicksort in parallel where each thread performs approximately 1.5 accesses per element. For merging, two memory accesses per element are required.
- Compressing consecutive ranges of the in-memory sorted buffer.
- Sending compressed ranges to the CNs.

$$T_{read} = \texttt{PFS\_LATENCY} + \frac{\texttt{SDN\_READ\_SIZE}}{\texttt{PFS\_BW}}$$

$$T_{sort} = \Bigg(1.5 \times \frac{\texttt{SDN\_READ\_SIZE}}{\texttt{CORES\_PER\_NODE}} log\left(\frac{\texttt{SDN\_READ\_SIZE}}{\texttt{TYPE\_SIZE} \times \texttt{CORES\_PER\_NODE}}\right) + 2 \times \texttt{SDN\_READ\_SIZE} + 2 \times \texttt{SDN\_BUFFER\_SIZE}\Bigg) \times \frac{1}{\texttt{MEMORY\_BW}}$$

$$T_{comp} = \frac{\texttt{SDN\_BUFFER\_SIZE}}{\texttt{COMPRESSION\_BW} \times \texttt{CORES\_PER\_NODE}}$$

$$T_{send} = \frac{\texttt{SDN\_BUFFER\_SIZE} \times \texttt{WCCR}}{\texttt{NETWORK\_BW}}$$

Based on the algorithm, T_{read} is overlapped with T_{sort}. Once the in-memory buffer is full (after $\frac{\texttt{SDN\_BUFFER\_SIZE}}{\texttt{SDN\_READ\_SIZE}}$ rounds of reading/sorting), compression happens. All of these operations are overlapped with T_{send}. Therefore, the time it takes for SDN from the moment it starts reading data to the moment it sends out messages to CNs (which we call SDN's gap) can be expressed as follows:

$$SDN_{gap}^{read} = max\left\{\frac{\texttt{SDN\_BUFFER\_SIZE}}{\texttt{SDN\_READ\_SIZE}} \times max\left\{T_{read}, T_{sort}\right\} + T_{comp}, T_{send}\right\}$$

The total effective execution time of each SDN in read phase is the sum of all these gaps for all communication rounds where SDN sends data to CNs. Hence, the total effective execution time of SDNs can be expressed as:

$$T_{SDN}^{read} = \frac{\texttt{FILE\_SIZE}}{\texttt{NUM\_SDN} \times \texttt{SDN\_BUFFER\_SIZE}} \times SDN_{gap}^{read}$$

One can repeat the same analysis for the other five components of our sorting algorithm and find the effective execution time of each component. Total execution time of the algorithm, then, can be expressed as follows:

$$T = max\left\{T_{SDN}^{read}, T_{CDN}^{read}, T_{CN}^{read}\right\} + max\left\{T_{SDN}^{write}, T_{CDN}^{write}, T_{CN}^{write}\right\}$$

4. EVALUATION

In this section we describe the characteristics of the testbed system of our choice, we provide an overview of microbenchmarks chosen to find critical system parameters, and finally we demonstrate tuning results of our implementation and performance comparison with BigSort algorithm (Integer sorting in CORAL benchmark [1]).

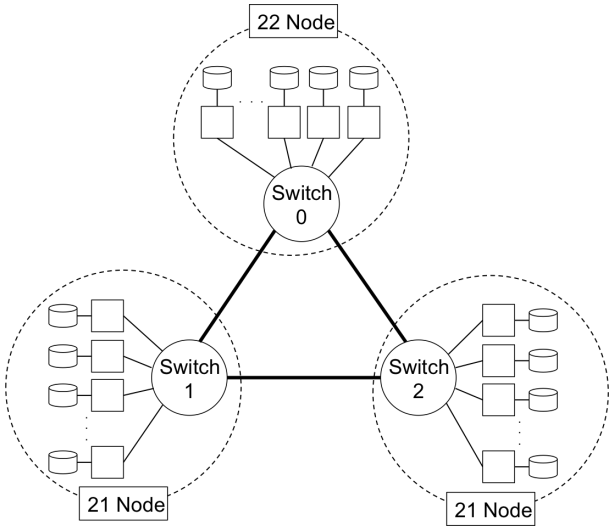


Figure 1: Topology of the experiment cluster.

4.1 System Setup

Our experiments were conducted on a 65-node SUN Fire Linux-based cluster, with one head node and 64 computing nodes. Each compute node was equipped with a quad-core AMD Opteron that supports up to 8 simultaneous integer threads. A selected number of nodes were also equipped with SSD along with HDD. Figure 1 depicts the topology of this cluster. We used the MPICH3.1.4 release and installed and deployed a PVFS2 parallel file system [25] with version OrangeFS 2.8.2.

The DEP system architecture assumes the close proximity of Storage-side Data Nodes (SDNs) to file-servers and also Compute-side Data Nodes (CDNs) to Compute Nodes (CNs). We chose the above testbed machine for several reasons, mostly because it provides us with certain capabilities: 1) all nodes are equipped with hard drives and some with SSDs making it easier to set specific nodes to represent different group of nodes (i.e. SDNs and CDNs), 2) the network of the cluster also allows us to mimic the DEP network topology such as high bandwidth between SDNs and file-servers or CDNs and CNs, and 3) much like current supercomputers, we are able to see the effect of the limited bandwidth between CNs and SDNs by selecting appropriate subsets of nodes.

Specifically, our experiment setup was as follows. We deployed a PVFS2 instance on 8 nodes on switch 0 using their HDD as the storage layer. We chose SDNs amongst available nodes of switch 0 exploiting the high intra-node bandwidth to the parallel file system, whereas on the other hand, we chose CDNs and CNs amongst nodes on switches 1 and 2 using the slower inter-node link.

In the end-to-end comparison we compared our sorting algorithm with BigSort. We used the same PFVS2 configuration for BigSort. However, we chose compute nodes running BigSort amongst nodes outside of switch 0, where PVFS2 was deployed, making our comparison as fair as possible.

4.2 Benchmarking and System Parameters

We ran a collection of micro-benchmarks to discover the values of the system parameters to better tune our implementation:

Table 1: System Model Parameters

Parameter	Value
PFS_LATENCY	2.8 ms
PFS_BW	111/162 MiB/s
SSD_LATENCY	5 us
SSD_BW	368 MiB/s
MEMORY_BW	2.8 GiB/s
NETWORK_BW	10.9 MiB/s
COMPRESSION_BW	520 MiB/s
DECOMPRESSION_BW	604 MiB/s
WCCR[2]	0.3-0.5

- **Latency of parallel file system.** This is achieved by measuring the time it takes to write/read a small portion to/from a random location of a file. We executed thousands of accesses and took the average.
- **Bandwidth to parallel file system.** This is achieved by running a micro-benchmark that uses MPI-IO collective operations to read/write a contiguous linear array with 128 MiB of data from/to parallel file system. Note that if we run the benchmark on nodes within the same switch as PVFS2, we get 162 MiB/s bandwidth, while if we run the benchmark from another switch the bandwidth is 111 MiB/s. In the later case limited network bandwidth between two switches causes a reduction of the effective IO throughput seen by the compute nodes. The result of this benchmark signifies even more the benefits of using SDNs in the DEP architecture. In general, it is expected that the SDNs get a higher bandwidth to parallel file system due to their proximity to file-servers.
- **Latency and bandwidth to local SSD.** Similarly, we ran the same micro-benchmarks but this time accessing the local SSD instead of the parallel file system.
- **Memory bandwidth.** To measure the main memory bandwidth of our testbed machine we ran STREAM [4] benchmark.
- **Compression and decompression bandwidth.** We ran the compression algorithm on 128 MiB of data (integers) to measure this parameter.

Table 1 contains the full list of the values of system parameters obtained by running these benchmarks.

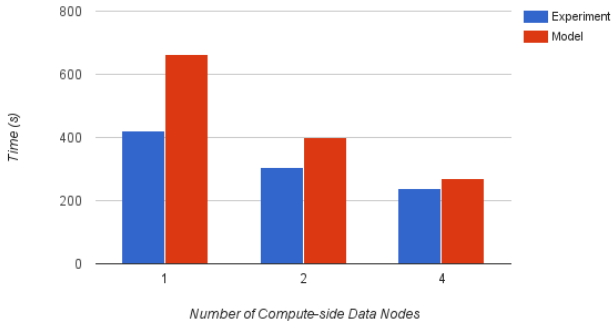
4.3 Performance Tuning

There are opportunities for performance optimization by tuning two parameters: the size of the various buffers used and the ratio of number of CNs to CDNs and SDNs. We used the performance model to get the optimal size of the buffers used in different types of nodes. This tuning considers the memory requirement of each type of node and the restriction that the total amount of memory usage in each node should not exceed the total available RAM (i.e. DRAM_ALLOCATION). Table 2 contains values of tunable and derived parameters obtained from performance model.

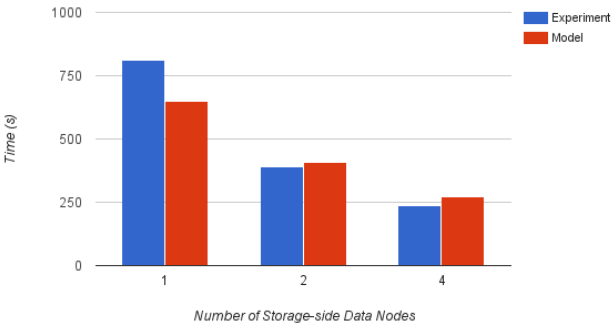
[2]Worst-case compression ratio

Table 2: Tunable and Derived Model Parameters

Parameter	Value
NUM_SDN	1,2,4
NUM_CDN	1,2,4
NUM_CN	4,8,12,16,20,24
CORES_PER_NODE	8
DRAM_ALLOCATION	1 GiB
SDN_BUFFER_SIZE[3]	32,64,128 MiB
SDN_READ_SIZE	16,32 MiB
CN_MERGE_BUFFER_R[4]	200 MiB
CHUNK_SIZE	16,20 MiB
CN_MERGE_BUFFER_W[5]	SDN_BUFFER_SIZE
TYPE_SIZE	4 bytes (integers)
FILE_SIZE	$2 \times (NUM\_SDN + NUM\_CDN + NUM\_CN)$ GiB
CN_RECV_BUFFER	$\frac{SDN\_BUFFER\_SIZE}{NUM\_CN}$
NUM_CHUNKS	$\frac{CDN\_MERGE\_BUFFER}{CHUNK\_SIZE}$
NUM_SERIES	$\frac{FILE\_SIZE}{NUM\_CN \times CN\_MERGE\_BUFFER\_R}$



(a) Effect of number of CDNs (with 4 CN, 4 SDN)



(b) Effect of number of SDN (with 4 CN, 4 CDN)

Figure 2: Effect of different number of CDNs and SDNs on performance. File size in all these experiments was 16 GiB.

Figure 2 shows the effect of varying number of CDNs/SDNs in performance. It is clear that with constant number of CNs, larger number of CDNs and/or SDNs leads to lower total execution time. In the extreme case where the number of CNs, CDNs, and SDNs are all equal, each SDN/CDN is responsible to serve exclusively one CN. This causes the minimum amount of contention on CDNs/SNDs, and maximizes the parallelization. However, in larger-scale experiments, CNs are the most numerous resources available in a system, hence matching up the number of CDNs and SDNs to number of CNs is not usually feasible. Therefore, for the rest of our experiments on our small-scale prototype testbed, we assume that the number of SDNs and also the number of CDNs is 4.

Note that the times derived from performance model are very close in almost all cases to actual execution times in different settings in Figure 2. This means we can simply use the performance model to predict the optimal number of CNs, CDNs, and SDNs in a different DEP cluster/setting and avoid running the algorithm (which may take rather long) for the sake of tuning.

4.4 Scalability Study

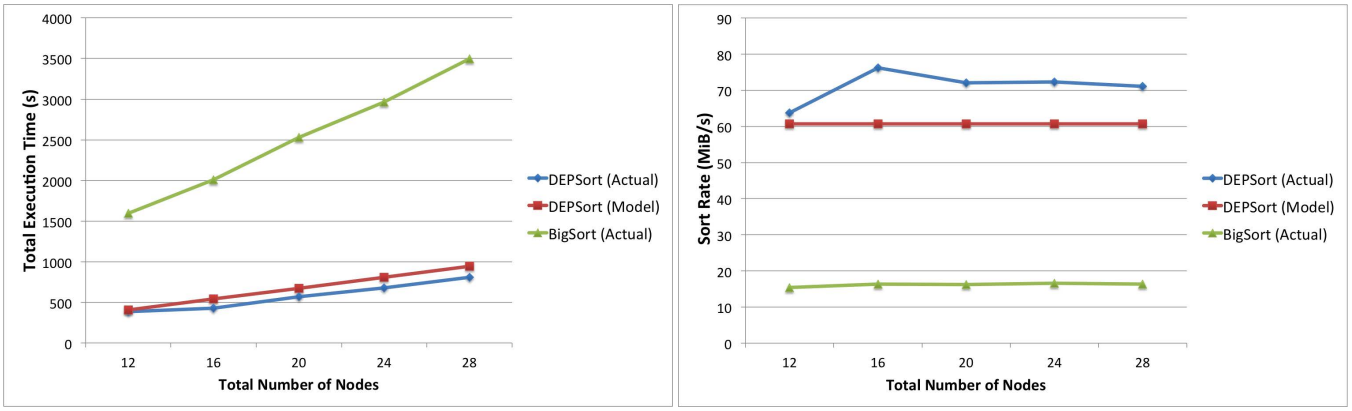
We compare our implementation with BigSort and Figure 3 depicts the weak scalability graph for execution time (3a) and sorting rate (3b). For each experiment, the file size is twice the aggregate memory allocation of the total number of nodes (i.e. for 12 nodes we used 24 GiB file size, for 16 nodes we used 32 GiB file size, and so on). To have a fair comparison, the total number of nodes reported in the graph for DEPSort is equal to the sum of number of CNs, CDNs, and SDNs. As mentioned before, we use 4 CDNs and 4 SDNs for DEPSort experiments. DRAM_ALLOCATION for both DEPSort and BigSort is set to 1 GiB.

First, the performance model matches almost perfectly with the timing from the actual experiments on the cluster. This means the performance model is accurate enough and can be used to predict the execution time of our sorting algorithm on a DEP cluster.

Second, the average rate at which our algorithm sorts the data is 71 MiB/s. Considering that bandwidth to PVFS2 from SDNs is 162 MiB/s, and also the fact that for sorting the data we have to at least read the entire raw data once and write the final sorted data back to the disk, the best achievable rate is 81 MiB/s. This means our sorting technique almost entirely overlaps computation and communication with just reading the raw data and writing back the sorted data. Note that the maximum achievable sort rate with a non-DEP implementation on the same cluster is 55 MiB/s (due to the 111 MiB/s bandwidth from CNs to PFVS2 mentioned in Table 1). This proves the effectiveness of DEP architecture where having SDNs closer to parallel file system improves the performance by at least 30% compared to the best possible sort algorithm in a non-DEP architecture.

Third, DEPSort is on average 4.4X faster than BigSort. This is due to the entire overlap of communication and computation with disk operation in DEPSort, and also the fact that DEPSort compresses the intermediate results and stores them in fast SSDs. In case of BigSort, the intermediate results are written back to the parallel file system, and also the actual sorting algorithm reads and writes the inter-

[3]Size of in-memory buffer
[4]Merge buffe size in read phase
[5]Merge buffe size in write phase



(a) Sort Execution Time

(b) Sort Rate

Figure 3: Weak scaling study of DEPSort and BigSort.

mediate data multiple times. The benefit of our algorithm in performance, thus, comes from two different reasons: 1) the algorithmic changes we proposed (overlapping computation with communication, compressing the intermediate results), and 2) from the DEP architecture itself (higher IO throughput through SDNs and fast SSD devices close to CNs).

Lastly, as expected, the performance of DEPSort gradually drops at larger number of nodes. This is mainly because of the contention over CDNs and SDNs. In the largest case, we have 20 CNs, 4 CDNs, and 4 SDNs. That is a ratio of 5 for the number of CNs over 1 CDN/SDN. Due to the resource limitations in our prototype DEP cluster we were not able to run the algorithm at larger scale and discover what the best ratio of CNs over CDNs/SDNs is for which the performance is still acceptable.

5. RELATED WORK

Data-intensive computing is already a reality in modern supercomputing sites. Sorting large number of data that cannot fit in memory (i.e. out-of-core) is just an example of the challenges we faced when dealing with data movement and manipulation. Disk-to-disk sorting is better explored in the distributed environments with TritonSort [23] leading in performance in the famous Graysort benchmarks [3]. Databricks utilized Apache Spark [30] to offer a competitive solution to the sorting problem.

However, in the HPC community where the machines are much more capable with sophisticated networks and specialized hardware, sorting was never a real problem until the BigData era. Sundar et al in [28] introduced a new out-of-core sort algorithm taking advantage of the asynchronous mechanisms present in modern supercomputers. By careful use of the available storage and a formulation of asynchronous data transfer, they were able to almost completely hide the computation behind the IO latency. In addition, Jose et al in [19] tried to demonstrate that even though Hadoop based solutions hold the record in sorting, MPI-based approaches can deliver similar performance. They presented a new hybrid out-of-core sorting algorithm that makes use of both MPI and OpenSHMEM PGAS model and they showed that their approach outperforms existing MPI-based implementation. However, our work introduces compression of the data before going through the slower interconnect network that minimizes even more the execution time and with our optimizations we were also able to almost entirely eliminate the IO of intermediate sort results with input and output giving our implementation an edge. Moreover, the DEP architecture with the CDNs, SDNs helped us further reduce the execution time by storing the *compressed* intermediate results closer to the CNs, and by accessing the parallel file system at a higher rate.

The use of additional specialized nodes is not a new idea. In [31], Zheng et al utilized different nodes in the IO path to selectively place data and perform analytics. They introduced a middleware that exploits the flexible placement of data to offer support to a variety of simulation and analytics workloads at large-scale. Similarly, in our previous work, we introduced a novel runtime system [14] that realizes the DEP architecture by decoupling the data-intensive phases and shipping them to run on specialized nodes such as SDNs and CDNs. This paper extends the understanding of the DEP architecture by presenting an out-of-core implementation of sorting and shows the high potentials of this architecture when dealing with data-intensive problems.

6. CONCLUSION

As the community is moving toward exa-scale era, certain challenges have to be addressed. One of the biggest challenges is the IO-wall problem where the computation rate is trending faster than data access rate. Amongst the mainstream solutions for this challenge, the DEP system architecture offers unique characteristics to significantly reduce the amount of data movement from storage layers to computational units.

In this paper we investigated potentials of DEP architecture for a well-known data-intensive kernel, disk-to-disk sorting. We designed an optimized algorithm in the context of DEP capable of sorting data-sets that cannot fit into aggregate available RAM. We presented a detailed performance model of our algorithm and showed that the model is a close match to reality. We demonstrated that our algorithm performs very close to the theoretically optimal sort algorithm. Advantages of our algorithm over other sorting algorithms in HPC are three-fold: higher IO throughput to parallel file system by accessing data through SDNs, data compression before communication over slower interconnect network, and

storing *compressed* intermediate data in SSD devices close to computational units. We also showed that our algorithm performs 30% faster than the theoretically optimal sort algorithm that is to run on the same testbed but is not designed to exploit the DEP architecture.

As part of our future work, we are planning to use the performance model and verify its accuracy even further by running our algorithm on a larger-scale newer DEP prototype. We also plan to implement and analyze another data-intensive application for the DEP architecture to expand our view on effectiveness of DEP. As an ultimate goal we would like to develop a simple-to-use programming model, similar to OpenMP and OpenACC, that lets user change legacy data-intensive codes minimally and yet offloads the data-intensive parts of the application to appropriate nodes of the DEP architecture.

7. ACKNOWLEDGMENTS

This work is supported by National Science Foundation Computer Systems Research under grants CNS-1162488, CNS-1162540, and CNS-1161507.

8. REFERENCES

[1] CORAL benchmark. asc.llnl.gov/CORAL-benchmarks.

[2] Graph500 benchmark. www.graph500.org.

[3] GraySort. sortbenchmark.org.

[4] STREAM benchmark. www.cs.virginia.edu/stream/.

[5] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *ACM SIGOPS Operating Systems Review*, volume 32, pages 81–91. ACM, 1998.

[6] R. Castain and O. Kulkarni. MapReduce and Lustre: Running Hadoop in a high performance computing environment. In *Intel Developers Forum*, volume 2013, 2013.

[7] C. Chen, Y. Chen, K. Feng, Y. Yin, H. Eslami, R. Thakur, X.-H. Sun, and W. D. Gropp. Decoupled I/O for data-intensive high performance computing. 2014.

[8] F. Chen, D. A. Koufaty, and X. Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *Proceedings of the international conference on Supercomputing*, pages 22–32. ACM, 2011.

[9] Y. Chen, C. Chen, X.-H. Sun, W. D. Gropp, and R. Thakur. A decoupled execution paradigm for data-intensive high-end computing. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 200–208. IEEE, 2012.

[10] A. Choudhary, W.-k. Liao, K. Gao, A. Nisar, R. Ross, R. Thakur, and R. Latham. Scalable I/O and analytics. In *Journal of Physics: Conference Series*, volume 180, page 012048. IOP Publishing, 2009.

[11] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with COTS HPC systems. In *Proceedings of the 30th international conference on machine learning*, pages 1337–1345, 2013.

[12] S. Conway. High performance data analysis: Big data meets HPC, March 2014. www.scientificcomputing.com [Online; posted 07-March-2014].

[13] J. Dongarra et al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, Feb. 2011.

[14] K. Feng, Y. Yin, C. Chen, H. Eslami, X.-H. Sun, Y. Chen, R. Thakur, and W. Gropp. Runtime system design of decoupled execution paradigm for data-intensive high-end computing. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–1. IEEE, 2013.

[15] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[16] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. High-performance design of HBase with RDMA over InfiniBand. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 774–785. IEEE, 2012.

[17] I. D. C. (IDC). First in-depth forecasts for worldwide HPC big data market, June 2014. http://www.idc.com [Online; posted 18-June-2014].

[18] N. S. Islam, M. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 35. IEEE Computer Society Press, 2012.

[19] J. Jose, S. Potluri, H. Subramoni, X. Lu, K. Hamidouche, K. Schulz, H. Sundar, and D. K. Panda. Designing scalable out-of-core sorting with hybrid MPI+ PGAS programming models. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 7. ACM, 2014.

[20] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, et al. Memcached design on high performance RDMA capable interconnects. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 743–752. IEEE, 2011.

[21] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[22] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao. A multiplatform study of I/O behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, New York, NY, USA, 2015. ACM.

[23] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced large-scale sorting system. In *NSDI*, 2011.

[24] R. Ross, R. Thakur, W. Loewe, and R. Latham. Parallel I/O in practice. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 216. ACM, 2006.

[25] R. B. Ross, R. Thakur, et al. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*, pages 391–430, 2000.

[26] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W.-K. Liao, and A. Choudhary. Enabling active storage on parallel I/O software stacks. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–12. IEEE, 2010.

[27] J. Sparks, H. Pritchard, and M. Dumler. The Cray framework for Hadoop for the Cray XC30.

[28] H. Sundar, D. Malhotra, and K. W. Schulz. Algorithms for high-throughput disk-to-disk sorting. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 93:1–93:10, New York, NY, USA, 2013. ACM.

[29] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. Ross, and K. Yoshii. Accelerating I/O forwarding in IBM Blue Gene/P systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–10. IEEE, 2010.

[30] R. Xin, P. Deyhim, A. Ghodsi, X. Meng, and M. Zaharia. GraySort on Apache Spark by Databricks. In *GraySort Competition*, 2014.

[31] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, et al. FlexIO: I/O middleware for location-flexible scientific data analytics. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 320–331. IEEE, 2013.