Design and Analysis of Disk-to-Disk Sorting in Decoupled Execution Paradigm

I. Introduction

Many scientific applications running in HPC systems deal with large amount of data [1–3]. Furthermore, Big Data analytics are not just limited to commodity clusters, since many use cases of these data analytic problems started to arise in HPC infrastructure as well [4–11].

On the other hand, the computation power of current multicore/manycore architectures has followed a faster trend compared to data access performance improvement (latency and bandwidth) [2, 12]. This gap is very unlikely to be overcome in the near future. Thus, accessing large amount of data becomes a bottleneck for many applications.

Current HPC architectures lack the support for users to control devices in the I/O stack, hence a data-intensive application that follows the existing architecture is forced to move all input data between the underlying storage solution and the compute nodes, passing through slower networks. This movement of large amount of data constantly in an application can cause severe degradation to the overall performance.

There has been work on proposing new architectures to alleviate this bottleneck by decoupling and shipping data intensive operations closer to data [13–16]. Despite the fact that some of these new system architectures for data-intensive computing change the execution model of an application, scientists still use the old-fashion compute-centric parallel programming models to program these systems [17–20]. These programming models primarily focus on the memory abstractions and communication mechanism among processes. I/O is treated as a peripheral activity and often a separate phase in these programming models, which is often achieved through a subset of interfaces such as MPI-IO. This mismatch of programming data-intensive applications with compute-centric approaches makes the programming difficult and leads to error prone and complex codes.

One of the promising system architectures for data-intensive computing is the Decoupled Execution Paradigm (DEP) [21, 22], where the application is decoupled into computational-intensive and data-intensive phases and each are directed to the appropriate capable nodes. Although this new execution paradigm is promising as is, it lacks a programming model to facilitate developers to make the most out of the system with minimal effort. Considering the fact that future generation of supercomputers will have programmable devices in their I/O stack, the idea of having a framework to facilitate this programming for end-users sounds appealing.

As the first step toward achieving a solid, easy to use programming model for DEP architecture, one has to come up with primitive execution patterns and constructs that are theoretically and practically appropriate for this architecture. In this project, we aim to design, analytically model, quantitatively analyze, and evaluate an important data-intensive application, **disk-to-disk sorting**, in the context of DEP architecture.

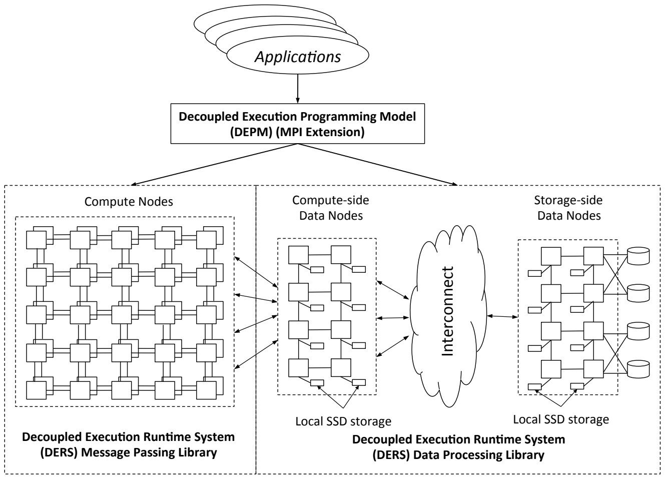

Applications
Decoupled Execution Programming Model (DEPM) (MPI Extension)
Compute Nodes
Compute-side Data Nodes
Storage-side Data Nodes
Interconnect
Local SSD storage
Local SSD storage
Decoupled Execution Runtime System (DERS) Message Passing Library
Decoupled Execution Runtime System (DERS) Data Processing Library

Fig. 1: Decoupled Execution System Architecture

II. Background

DEP is the first paradigm enabling users to identify and handle data-intensive operations separately. It can significantly reduce costly data movements and is more comprehensive than the existing execution paradigms for data-intensive applications. DEP decouples the nodes into data-processing (data) nodes and compute nodes. Data-processing nodes are further decoupled into compute-side data nodes and storage-side data nodes. Compute-side data nodes are compute nodes that are dedicated for data processing. Storage-side data nodes are specially designed nodes that are connected to storage devices with fast network. Compute-side data nodes reduce the size of computing generated data before sending it to storage nodes. Storage-side data nodes reduce the size of data retrieved from storage before sending it to compute-side data nodes. Data nodes can provide simple data forwarding without any data size reduction, but the idea behind data nodes is to let the data nodes conduct the decoupled data-intensive operations and optimizations to reduce the data size and movement. Figure 1 depicts the DEP architecture.

III. Design

Our design of disk-to disk sorting was driven by the need to utilize all the different parts of the DEP architecture, the Storage-side Data Nodes (SDNs) –sometime we interchange the name with simply file-server–, Compute Nodes (CNs) and Compute-side Data Nodes (CDNs) – sometime we interchange the name with simply I/O forwarders–. SDN are connected to the filesystem, 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 filesystem. CDNs, being connected to fast local SSDs, will be responsible for storing intermediate results. Finally, the compute nodes, 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 being distributed to the compute nodes, which perform some initial processing and send the data for intermediate storage to the I/O forwarders. 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. CNs then compute the final result and send the sorted data to SDNs for storage.

The following sections provide a more detailed description of each node type's operation.

A. *Storage-side Data Node (SDN)*

During the read phase, SDNs read data from the input file and sort them until a certain number of data is gathered and sorted. Then, they will distribute the data to each CN assigning a specific range to each CN. This is repeated until all the input data are sent, in which case a SDN will send a finisher to all CNs notifying them about the end of the incoming data.

SDNs maintain a read buffer, a merge buffer of the same size and two merge buffers, mem_sort and mem_merge. They read data from the input file into the read buffer and sort them into the merge buffer using parallel quicksort. Then, the mem_sort buffer, which contains already sorted data, is merged with the merge buffer into the mem_merge buffer, swapping mem_merge and mem_sort between consecutive merges, until mem_merge buffer is full. When this is reached, the SDN splits the mem_merge buffer into smaller buffers that contain data belonging to each CN. The range of data assigned to a CN depends on its rank in the global communicator, thus the ranges are known to the SDN and splitting the merge buffer only requires determining the index where each range begins. The buffers for each CN are then compressed and sent. An additional buffer is required to store the result of the compression of the mem_merge buffer.

Since the input is a single file, SDNs use MPI-IO to open, read, and close the file. In order to overlap file reading with sorting and merging, SDNs use double buffering, using MPI_File_iread_at maintaining an additional read buffer to swap between consecutive iterations. Both sorting the read data and compressing the data assigned to each CN are parallel using OpenMP.

After all the input data are sent to CNs, SDNs finish the read phase and proceed to the write phase, in which they mainly wait for messages received from CNs, decompress the message and write it to the output file at the appropriate location.

At the beginning of the write phase, each SDN computes the CNs that are assigned to it based on its own as well as their ranks. Then, it waits for a message from each one of them, containing the total number of elements that were sent to each CN for sorting. The total number of elements sent to their assigned CN can be used to compute the total number of elements that this SDN will write to the output file. Also, due to the fact that each CN is assigned to handle a specific range and that the CNs assigned to a SDN has consecutive ranks in global communicator, the ranges assigned to them are also consecutive, forming a single range that a SDN is responsible for writing to the output file. As soon as all the SDNs receive all the messages from their assigned CNs, they compute the number of elements that they need to write to the output file and then perform an exclusive scan on this value, after which the offset where they should start is computed. Then, using this offset as a start offset, each SDN can compute the offset at which the data from each CN need to be written by performing another exclusive scan with the total number of elements per CN.

Now that the file offset for each CN is computed, the SDNs are ready to receive data, and post receive requests from each of their assigned CNs. When a receive is completed, the SDN decompresses the received data, finds the file offset for the source CN, writes the decompressed data into the file, increases the file offset according to the size of the received data and posts a new receive from the source CN. The offset is updated this way due to the fact that the data sent by a CN are consecutive in the output and are only split-ed due to memory restrictions. If the received message was empty, that identifies the end of data from this CN and therefore no receive is posted. The SDN write phase is complete when all CNs assigned to it are done sending data.

The SDNs use double buffering to overlap receiving a message with decompressing and file writing, and additionally use asynchronous file writing to overlap file writing with waiting for the next message to arrive.

The SDN write phase requires only three buffers that can fit the message sent by the CNs during their write phase, two of them is of the size of compressed message that arrives.

B. *Compute Node (CN)*

During the read phase, the CNs are receiving sorted and compressed data from all the SDNs. Each CN maintains a buffer of sorted data that it has already received. Upon receiving, the new data is decompressed and merged with the local buffer, and now constitute this CN's sorted data. This process is repeated until the local buffer is not big enough for the already sorted data in memory and the new data to be merged into it. When this occurs, the CN needs to flush its local data to its corresponding CDN for intermediate storage. Each CN corresponds to a single CDN, which is determined by rank. The CN splits the local buffer into smaller chunks of data, which are then compressed and sent to the corresponding CDN. Finally, an empty message is sent to the CDN, to signal that there are no more chunks of this local buffer to be received. The reason for splitting the data to smaller chunks is associated with the DRAM requirements during the write phase and will be explained later. Each local buffer is a series generated from this CN. Once the data is flushed to the CDNs, the local merge buffer becomes empty. The received data that triggers the end of the previous series is the first part of the new data series and will be copied into the local buffer. This process is repeated until a finisher (a message of size zero, containing no data) is received. Upon receiving the finisher, the local buffer of the CN is split-ed, compressed and sent to the corresponding CDN as the last data series of this CN. Finally, a finisher is sent to the corresponding CDN to signal the end of data from this CN. During this phase, each CN keeps track of how many elements in total it has received, by uprating this number at the end of each data series. This is required for a later phase, and will be explained at that time.

The data that each CN is receiving belong to a specific range, which depends on its rank in the global communicator. Assuming uniform distribution of the input data, each CN will be responsible for roughly the same amount of data, therefore we achieve good load balancing.

Now we will move on to the implementation of CN read phase. The CNs are using double buffering and MPI_Irecv to overlap receiving of new data with decompressing the received data and merging it with the local buffer. When the end of a data series is reached and the local buffer needs to be split in chunks which are then compressed, the compression of each chunk happens in parallel with OpenMP. Sending the compressed chunks to the CDN is also non-blocking, so that the CN can begin processing the next data series. However, it is not done in parallel, since the chunks need to be sent in a specific order, from first to last so that the data remain sorted. The CN will wait on the send requests posted for a data series when it reached to the point of splitting and compressing the next series, to make sure that the buffers to compress the new chunks into will be available. The last optimization performed at the CNs is during the beginning of a new data series. Instead of copying the decompressed message that triggered the end of previous data series into the local buffer that is input to the merging, an extra buffer of this size is maintained at the CN and will be passed as an input to the merging.

We will continue with the memory requirements of the CN read phase. The CNs maintain a received decompressed data buffer where the decompression result is stored and two buffers for merging local and received data, one of which is input and the other is the output of merging. Also, they need two receive buffers for performing double buffering, a buffer to store the compressed chunks, and an additional decompress buffer storing the received message that triggers the end of a series.

After a CN notifies its corresponding CDN for the end of incoming data, it begins its write phase. Initially, it sends the total number of elements assigned to it to its corresponding SDN, determined by rank, which is required to compute the offset where data from this node will be written.

During this phase, the CNs request the intermediate data from their corresponding CDN. Each CN needs a chunk per data series, and merges the data from all the series into the final sorted data. When the merge buffer is full, it compresses and sends it to the corresponding SDN, and continues merging the remaining data into the now empty merge buffer. If the data contained in a chunk of a series is finished, the CN sends a request containing the series identifier and waits until receiving the next chunk of this series, decompresses it and continues the merging. If the received message is a finisher for a series, this indicates that there are no more data from this data series and it can be ignored for the remainder of the write phase.

As an optimization, the CNs use double buffering in two cases. First, so that the decompression and merging can be overlapped with the receive. Two buffers that can fit a compressed chunk are maintained per data series, and as soon as a chunk is decompressed and it is determined that it is not a finisher, the buffers are swapped and a non-blocking receive for the next chunk is posted. Second, so that the sending of the compressed merge buffer can be overlapped with the other operations. Two buffers of the same size are maintained, merge and send buffer, and as soon as the merge buffer is full, they are swapped. The send buffer is then compressed and sent to the SDN, while merging continues to the merge buffer.

Finally, we will discuss the memory requirements of the CN write phase. It requires two receive buffers for a compressed chunk and a buffer for a chunk for each data series, two buffers for merging and sending and an additional buffer for compressing the send buffer. As depicted by this description, the chunk size affects the memory requirements in a significant way, since it is multiplied by the number of data series per CN which increases with the size of the input file. This is the reason why the data series needed to be split-ed in smaller chunks during the read phase: all the data series generated by a CN would not fit in its available DRAM for merging into the final result, as this is the requirement for the write phase's memory allocation.

C. *Compute-side Data Node (CDN)*

During the read phase, the CDNs will be responsible for storing intermediate results that are received from all the CNs that are assigned to them, which is specified by their ranks in the communicator consisting of CNs and CDNs.

Each CDN maintains a local file where the received messages are stored, and a data structure indexed by a source CN, a data series and a chunk id, which is the number of the chunk in the chunk sequence created after spitting the data series in the CN. This structure returns the offset into the local file where this chunk of the specified data series of the specified CN is located and its size. Additionally, the CDN maintains a flag per CN that shows whether a data series finisher was the last message received by this CN, and a counter that keeps track of how many CNs have no more data to send.

Each CDN expects at least one message from each of its assigned CNs, due to our assumption for uniform distribution of input data. Therefore, it begins by posting a non blocking receive request for each of its assigned CNs. Upon completion of one of them, it retrieves information about the source to identify the CN to which this chunk belongs, and using its data structure identifies the series this chunk belongs to and the chunk id, which is always the next id of the last already received chunk from this CN. A new receive is posted for the next chunk in the current series from the source CN, the received chunk is then written into the local file and the data structure is updated. Upon receiving an empty message, the CDN checks the data series finisher flag for the source CN. If it was not set, it sets it since the finisher for the current data series was just received, and a new receive is posted for the first chunk in the next series from the source CN. If it was already set, the finisher message means that there is no additional series from the source CN. In this case, the source CN is done sending data, and therefore no receive is posted and the counter is updated.

The CDN's data structure to keep track of the contents of the local file containing the intermediate results is implemented as a three dimensional array, indexed by CN, data series and chunk id, with each element being a pair of (chunk size, file offset).

The CDNs are using double buffering to overlap receiving a chunk and writing an already available chunk to the local file. They maintain a buffer per assigned CN where the corresponding posted MPI_Irecv is expected to place the next received chunk, and an additional buffer of the same size whose contents are written to the file. Upon receiving a new chunk, and since the file write is blocking, it is safe to swap the file write buffer with the receive buffer of the CN whose chunk was just received, post a new MPI_Irecv from that CN to the corresponding receive buffer and continue with the file write from the write buffer.

Moving on to the memory requirements of the CDN read phase, CDNs need to maintain a buffer large enough to fit a compressed chunk per assigned CN and an additional one for writing to the file.

As soon as all the CNs are done sending data, as determined by the maintained counter, the write phase of the CDN begins. During this phase, the CDNs send compressed sorted data at the granularity of chunks back to the CNs to compute the final result. In general, the CDNs wait for messages from their assigned CNs containing the series from which they request the next chunk, and maintain an array of counters indexed by CN and by data series that show which is the next chunk to be sent to the specified CN from the specified data series. Upon receiving a message, they identify the source as the CN requesting a chunk and the data series from the message content, and using their counters they identify which chunk is to be sent. The CDN accesses the data structure to determine if such a chunk exists. If it does, the CDN retrieves its file offset and size, reads it from the local file, sends it to the CN, increase the counter to point to the next chunk and posts a new receive request from the CN that was the source of the last received message, waiting for the next data series identifier. If it does not exist, the CDN sends a finisher message instead, to show that there are no more chunks from this data series. The CDN keeps track of the finisher messages it sends, to determine which CNs have received all of their data. When the last finisher is sent, the CDN write phase is complete.

As an optimization, an CDN begins by sending the chunks with id zero from all data series to its assigned CNs. After these sends has been completed, it also sends the chunks with id one from all data series or a series finisher if there is no such chunk for a data series. These messages will always be requested by the CNs, and thus there is no need to wait for explicit requests from the assigned CNs for them. Additionally, the CDN uses double buffering to overlap reading a chunk from the local file with sending the previous chunk to a CN.

IV. PERFORMANCE MODELING

A. Model Parameters

Our performance model is defined based on a number of parameters. First, there are parameters describing the system we are running on. These are:

- NUM_FS: Number of SDNs.
- NUM_IO: Number of CDNs.
- NUM_CN: Number of CNs.
- CORES_PER_NODE: Number of cores per node.
- PFS_LATENCY: Latency of parallel filesystem.
- PFS_BW: Bandwidth of parallel filesystem.
- SSD_LATENCY: Latency of SSD.
- SSD_BW: Bandwidth of SSD.
- STREAM_BW: STREAM Copy Bandwidth.
- NETWORK_BW: Network Bandwidth.
- DRAM_ALLOCATION: Available memory per node.
- COMPRESSION_RATE: Rate achieved by the compression.
- DECOMPRESSION_RATE: Rate achieved by the decompression.
- FILE_SIZE: Size of input file in bytes.
- TYPE_SIZE: Size of type of data to be sorted in bytes.
- WCCR: Worst case ratio of compressed to initial size achieved by compression.

The next set of parameters describes the sizes of the buffers and/or the message sizes that are used throughout the different phases of the algorithm, and which we can tune to achieve better performance. These are:

- FS_BUFFER_SIZE: Size of merge buffer, in SDN read phase.
- FS_READ_SIZE: Size of data read from file, in SDN read phase.
- CN_MERGE_BUFFER: Size of merge buffer, in CN read phase.
- CHUNK_SIZE: Size of chunk in bytes, in CN read phase.
- CN_MMERGE_BUFFER: Size of merge buffer, in CN write phase.

Finally, based on these parameters, we define a set of derived parameters that are useful in expressing our model. These are:

- PFS_RATE: Rate of parallel filesystem, computed as the inverse of PFS_BW.
- SSD_RATE: Rate of SSD, computed as the inverse of SSD_BW.
- STREAM_RATE: Rate of STREAM Copy, inverse of STREAM_BW.
- NETWORK_RATE: Rate of network, computed as the inverse of NETWORK_BW.
- CN_RECV_BUFFER: Size of receive buffer, in CN read phase. Assuming uniform distribution of data, each SDN will distribute its buffer, of size FS_BUFFER_SIZE, equally amongst the CNs. Allowing for some imbalance, we set this size to $1.5 \times \frac{\texttt{FS\_BUFFER\_SIZE}}{\texttt{NUM\_CN}}$.
- NUM_CHUNKS: Maximum number of chunks per merge buffer, in CN read phase, computed as $ceiling\left(\frac{\texttt{CN\_MERGE\_BUFFER}}{\texttt{CHUNK\_SIZE}}\right)$
- NUM_SERIES: Number of data series generated by a CN. Assuming uniform distribution of data, each CN will be assigned roughly $\frac{\texttt{FILE\_SIZE}}{\texttt{NUM\_CN}}$ bytes of data, and CN_MERGE_BUFFER bytes will constitute a series. Therefore, $\texttt{NUM\_SERIES} = \frac{\texttt{FILE\_SIZE}}{\texttt{NUM\_CN} \times \texttt{CN\_MERGE\_BUFFER}}$.

B. Constraints

The tunable parameters we defined in section IV-A are subject to some constraints, which are imposed due to the memory availability in the system. As described in section III, each phase of the algorithm has specific memory requirements. Expressing these using our model parameters will give us the constraints they need to satisfy.

According to section III-A, during the read phase a SDN requires three buffers of size FS_READ_SIZE, two of size FS_BUFFER_SIZE, and one of size FS_BUFFER_SIZE $\times$ WCCR, assuming the worst case. Therefore, we can express the constraint

$$3 \times \texttt{FS\_READ\_SIZE} + (2 + \texttt{WCCR}) \times \texttt{FS\_BUFFER\_SIZE} \\ \leq \texttt{DRAM\_ALLOCATION}$$

Additionally, during the write phase a SDN requires one buffer of size CN_MMERGE_BUFFER and two of size CN_MMERGE_BUFFER × WCCR, assuming the worst case. Therefore, we can express the constraint

$$(1 + 2 \times \texttt{WCCR}) \times \texttt{CN\_MMERGE\_BUFFER} \\ \leq \texttt{DRAM\_ALLOCATION}$$

According to section III-B, during the read phase a CN requires two buffers of size CN_MERGE_BUFFER, one of size CN_MERGE_BUFFER × WCCR, two of size CN_RECV_BUFFER and two of size CN_RECV_BUFFER × WCCR, assuming the worst case. Therefore, we can express the constraint

$$(2 + \texttt{WCCR}) \times \texttt{CN\_MERGE\_BUFFER} + \\ 2 \times \texttt{CN\_RECV\_BUFFER}(1 + \texttt{WCCR}) \\ \leq \texttt{DRAM\_ALLOCATION}$$

During the write phase a CN requires two buffers of size CN_MMERGE_BUFFER, one of size CN_MMERGE_BUFFER × WCCR. Additionally, it requires one of size CHUNK_SIZE and two of size CHUNK_SIZE × WCCR for each data series, NUM_SERIES. The corresponding constraint is

$$(2 + \texttt{WCCR}) \times \texttt{CN\_MMERGE\_BUFFER} + \\ \texttt{NUM\_SERIES} \times (1 + 2 \times \texttt{WCCR}) \times \texttt{CHUNK\_SIZE} \\ \leq \texttt{DRAM\_ALLOCATION}$$

Finally, according to section III-C, during the read phase an CDN requires a buffer of size CHUNK_SIZE × WCCR per CN that is assigned to it and an additional buffer of the same size for writing to the local file. The constraint is as follows.

$$\left(\frac{\texttt{NUM\_CN}}{\texttt{NUM\_IO}} + 1\right) \times \texttt{CHUNK\_SIZE} \times \texttt{WCCR} \\ \leq \texttt{DRAM\_ALLOCATION}$$

During the write phase, the maximum memory that an CDN requires is a buffer of size CHUNK_SIZE × WCCR per CN assigned to it. The corresponding constraint is

$$\frac{\texttt{NUM\_CN}}{\texttt{NUM\_IO}} \times \texttt{CHUNK\_SIZE} \times \texttt{WCCR} \\ \leq \texttt{DRAM\_ALLOCATION}$$

C. *Performance model*

In computing the time estimates due to message sending and receiving, we will ignore the latency of the network due to the fact that the messages that we are exchanging are large and thus the effect of the latency term ie negligible.

1) ***SDN Read Phase:*** During this phase, a SDN reads a buffer from the input file, sorts and merges into a merge buffer until it is full. It then splits the data amongst the CNs, compresses and sends the compressed data. We compute the following time estimates:

- $T_{FileRead}$: Time to read a buffer of size FS_READ_SIZE from the filesystem.
- $T_{SortMerge}$: Time to sort a buffer of size FS_READ_SIZE and merge with a buffer of size FS_BUFFER_SIZE. For the sort, we use CORES_PER_NODE threads to sort parts of the buffer using quicksort, which performs 1.5 accesses per element, and merge them together at the end. For the merge, two memory accesses are required per element assuming it fits in cache.
- $T_{Compression}$: Time to compress smaller ranges in a buffer of size FS_BUFFER_SIZE. The time of compression depends on the size of the data, thus we can use the size of the buffer to estimate it.
- T_{Send}: Time to send the compressed smaller ranges in a buffer of size FS_BUFFER_SIZE to the CNs.

$$T_{FileRead} = \texttt{PFS\_LATENCY} + \\ \texttt{FS\_READ\_SIZE} \times \texttt{PFS\_RATE}$$

$$T_{SortMerge} = (1.5 \times \frac{\texttt{FS\_READ\_SIZE}}{\texttt{CORES\_PER\_NODE}} \\ log\left(\frac{\texttt{FS\_READ\_SIZE}}{\texttt{TYPE\_SIZE} \times \texttt{CORES\_PER\_NODE}}\right) + \\ 2 \times \texttt{FS\_READ\_SIZE} + \\ 2 \times \texttt{FS\_BUFFER\_SIZE}) \times \texttt{STREAM\_RATE}$$

$$T_{Compression} = \frac{\texttt{FS\_BUFFER\_SIZE} \times \texttt{COMPRESSION\_RATE}}{\texttt{CORES\_PER\_NODE}}$$

$$T_{Send} = \texttt{FS\_BUFFER\_SIZE} \times \texttt{WCCR} \times \texttt{NETWORK\_RATE}$$

Due to the double buffering optimization, file reading is overlapped with sorting and merging, and then the compression is done. All of these operations are overlapped with sending a previously compressed merge buffer. Therefore, the time between the completion of the operations to prepare and send a merge buffer, which we will call $FS_{ReadGap}$, can be estimated using teh following formula.

$$FS_{ReadGap} = max\{ \\ \frac{\texttt{FS\_BUFFER\_SIZE}}{\texttt{FS\_READ\_SIZE}} \times max\{T_{FileRead}, T_{SortMerge}\} \\ + T_{Compression}, T_{Send}\}$$

2) ***CN Read Phase:*** During this phase, a CN receives data from the SDNs, decompresses and merges into a merge buffer until it is full. It then splits the merge buffer into chunks of data, compresses and sends them to an CDN, and repeats until no more data is received. We compute the following time estimates:

- $T_{Receive}$: Time to receive a compressed message sent by a SDN. The size of this message will be at most CN_RECV_BUFFER × WCCR based on our assumption about uniform data distribution. In this estimate, we need to account for the time that the SDNs need to compute and send these messages, i.e. the $FS_{ReadGap}$ as defined earlier. Since a CN is expected to receive messages from all SDNs during this phase, we can assume that the effective delay due to the $FS_{ReadGap}$ is estimated by $\frac{FS_{ReadGap}}{\texttt{NUM\_FS}}$. This time includes the message transfer time trough the network, thus we do not need to include it in the CN side.

- $T_{Decompression}$: Time to decompress a received message in a buffer of size CN_RECV_BUFFER.
- T_{Merge}: Time to merge two buffers into a result of size at most CN_MERGE_BUFFER. Two memory accesses per element are required for the merge assuming it fits in cache.
- $T_{Compression}$: Time to compress a merge buffer split into chunks. The time of compression depends on the size of the data, thus we can estimate it using the size of the buffer we need to compress.
- T_{Send}: Time to send all compressed chunks generated by a merge buffer to an CDN.

$$T_{Receive} = \frac{FS_{ReadGap}}{\texttt{NUM\_FS}}$$

$$T_{Decompression} = \texttt{CN\_RECV\_BUFFER} \times \texttt{DECOMPRESSION\_RATE}$$

$$T_{Merge} = 2 \times \texttt{CN\_MERGE\_BUFFER} \times \texttt{STREAM\_RATE}$$

$$T_{Compression} = \frac{\texttt{CN\_MERGE\_BUFFER} \times \texttt{COMPRESSION\_RATE}}{\texttt{CORES\_PER\_NODE}}$$

$$T_{Send} = \texttt{CN\_MERGE\_BUFFER} \times \texttt{WCCR} \times \texttt{NETWORK\_RATE}$$

Due to the double buffering optimization, message receiving is overlapped with decompressing and merging, and after the merge buffer is full the compression is done. All of these operations are overlapped with sending the compressed chunks of the previous merge buffer. Therefore, the time between the completion of the operations to prepare and send the chunks of a merge buffer, which we will call $CN_{ReadGap}$, can be estimated using the following formula.

$$CN_{ReadGap} = max\{\texttt{NUM\_CHUNKS} \times max\{T_{Receive}, T_{Decompression} + T_{Merge}\} + T_{Compression}, T_{Send}\}$$

*3) **CDN Read Phase**:* During this phase, an I.O forwarder receives data from the CNs and stores them in a local file. We compute the following time estimates:

- $T_{Receive}$: Time to receive a compressed chunk sent by a CN. The size of this message will be at most CHUNK_SIZE × WCCR based on its construction. In this estimate, we need to account for the time that the CNs need to compute and send these chunks, i.e. the $CN_{ReadGap}$ as defined earlier. Since an CDN is expected to receive messages from all the CNs that are assigned to it during this phase and that each CN sends roughly NUM_CHUNKS chunks, we can assume that the effective delay due to the $CN_{ReadGap}$ is estimated by $\frac{CN_{ReadGap} \times \texttt{NUM\_IO}}{\texttt{NUM\_CHUNKS} \times \texttt{NUM\_CN}}$. This time includes the message transfer time trough the network, thus we do not need to include it in the CDN side.
- $T_{FileWrite}$: Time to write the received message to a local file.

$$T_{Receive} = \frac{CN_{ReadGap} \times \texttt{NUM\_IO}}{\texttt{NUM\_CHUNKS} \times \texttt{NUM\_CN}}$$

$$T_{FileWrite} = \texttt{SSD\_LATENCY} + \texttt{CHUNK\_SIZE} \times \texttt{WCCR} \times \texttt{SSD\_RATE}$$

Due to the double buffering optimization, message receiving is overlapped with file writing, thus the time between writing consecutive received chunks to the local file is estimated by the formula

$$IO_{ReadGap} = max\{T_{Receive}, T_{FileWrite}\}$$

*4) **CDN Write Phase**:* During this phase, an I.O forwarder waits upon requests from the CNs, reads the requested compressed chunk from the local file and sends it. We compute the following time estimates:

- $T_{FileRead}$: Time to read a compressed chunk from the local file. This size is limited by $CHUNK\_SIZE \times WCCR$.
- T_{Send}: Time to send the read compressed chunk to a CN.

$$T_{FileRead} = \texttt{SSD\_LATENCY} + \texttt{CHUNK\_SIZE} \times \texttt{WCCR} \times \texttt{SSD\_RATE}$$

$$T_{Send} = \texttt{CHUNK\_SIZE} \times \texttt{WCCR} \times \texttt{NETWORK\_RATE}$$

Due to the double buffering optimization, message receiving is overlapped with file writing. Additionally, each CN will be expected to receive roughly $\frac{NUM\_CN}{NUM\_IO} \times NUM\_SERIES \times NUM\_CHUNKS$, so it is safe to assume that there will always be an outstanding request after a file read of a chunk is done. Therefore, the time between sending two consecutive read compressed chunks is estimated by the formula

$$IO_{WriteGap} = max\{T_{FileRead}, T_{Send}\}$$

*5) **CN Write Phase**:* During this phase, a CN receives compressed chunks from all data series, decompresses them and merges into a buffer, which once full is compressed and sent to a SDN. The following time estimates model the time required to compute and send a full merge buffer to a fileserver:

- $T_{Receive}$: Time to receive enough data that would fill the merge buffer after being decompressed. In this estimate, we need to account for the time that the CDN need to read and send these chunks, i.e. the $IO_{WriteGap}$ as defined earlier. Since an CDN is expected to send one passage per $IO_{WriteGap}$ time, a CN would wait for $\frac{NUM\_CN}{NUM\_IO} \times IO_{WriteGap}$ to receive a compressed chunk, and $\frac{NUM\_CN}{NUM\_IO} \times \frac{CN\_MMERGE\_BUFFER}{CHUNK\_SIZE} \times IO_{WriteGap}$ to receive all data to fill a merge buffer. This time includes the message transfer time trough the network, thus we do not need to include it in the CN side.
- $T_{Decompression}$: Time to decompress data up to the size of CN_MMERGE_BUFFER.
- T_{Merge}: Time to merge the received decompressed chunks into the merge buffer, up to the size of CN_MMERGE_BUFFER.

- $T_{Compression}$: Time to compress the computed merge buffer.
- T_{Send}: Time to send the compressed merge buffer to a SDN.

$$T_{Receive} = \frac{\texttt{NUM\_CN}}{\texttt{NUM\_IO}} \times \frac{\texttt{CN\_MMERGE\_BUFFER}}{\texttt{CHUNK\_SIZE}} \times IO_{WriteGap}$$

$$T_{Decompression} = \texttt{CN\_MMERGE\_BUFFER} \times \texttt{DECOMPRESSION\_RATE}$$

$$T_{Merge} = 2 \times \texttt{STREAM\_RATE} \times \texttt{CN\_MMERGE\_BUFFER}$$

$$T_{Compression} = \texttt{CN\_MMERGE\_BUFFER} \times \texttt{COMPRESSION\_RATE}$$

$$T_{Send} = \texttt{CN\_MMERGE\_BUFFER} \times \texttt{WCCR} \times \texttt{NETWORK\_RATE}$$

Due to double buffering, message receiving, message sending, and decompression-merging-compression are overlapped. Therefore, the time between sending two consecutive compressed merge buffers is estimated by the formula

$$CN_{WriteGap} = max\{T_{Receive}, T_{Decompression} + T_{Merge} + T_{Compression}, T_{Send}\}$$

6) ***SDN Write Phase****:* During this phase, a SDN receives messages from CNs, decompresses them and writes the final result into the file. The following time estimates model the time required to write a full merge buffer as received by a CN to a SDN:

- $T_{Receive}$: Time to receive a compressed merge buffer from a CN. In this estimate, we need to account for the time that a CN needs to compute and send such a buffer, i.e. the $CN_{WriteGap}$ as defined earlier. Since $\frac{NUM\_CN}{NUM\_FS}$ CNs are assigned to a single SDN, a SDN would wait for $\frac{NUM\_FS}{NUM\_CN} \times CN_{WriteGap}$ to receive a compressed merge buffer. The time for the message to be transferred via the network has been accounted for in the $CN_{WriteGap}$, thus it will not be included here.
- $T_{Decompression}$: Time to decompress the received merge buffer.
- $T_{FileWrite}$: Time to write the decompressed merge buffer to the output.

$$T_{Receive} = \frac{\texttt{NUM\_FS}}{\texttt{NUM\_CN}} \times CN_{WriteGap}$$

$$T_{Decompression} = \texttt{CN\_MMERGE\_BUFFER} \times \texttt{DECOMPRESSION\_RATE}$$

$$T_{FileWrite} = \texttt{PFS\_LATENCY} + \texttt{CN\_MMERGE\_BUFFER} \times \texttt{PFS\_RATE}$$

Due to double buffering, message receiving and decompressing is overlapped with file writing, thus the time between writing two consecutive merge buffers into the output file is estimated by the formula

$$FS_{WriteGap} = max\{T_{Receive} + T_{Decompression}, T_{FileWrite}\}$$

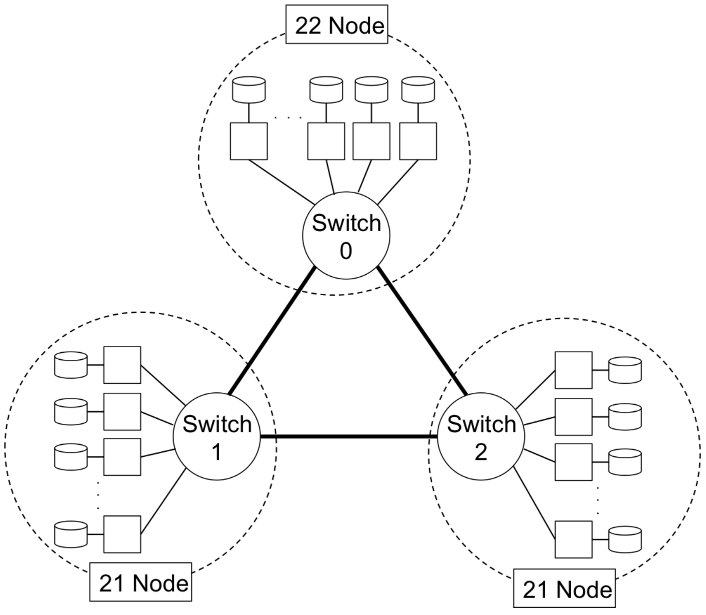


Fig. 2: Topology of the experiment cluster.

V. Experiments

In this section we describe our choice of infrastructure to run our implementation, give an overview on micro-benchmarks chosen to find critical system parameter, and demonstrate tuning and performance results of our implementation.

A. System Setup

Since our design is based on DEP system architecture, to get the most benefit out of our design, we have to choose an HPC testbed that provides certain capabilities. DEP architecture promises the close proximity of Storage-side Data Nodes (SDNs) to file-servers. Also, we based our design on availability of SSD in a few nodes close to Compute Nodes (CNs). Furthermore, in order to see the benefit of data reduction in our design, the testbed should provide a limited bandwidth between SDNs and CNs. We chose to run our implementation on IIT-HEC data-intensive HPC cluster. Figure **??** depicts the topology of this cluster. Each node is equipped with a Quad-core AMD Opteron 1.6Ghz with 4MB of L3 cache that supports up to 8 independent integer threads working simultaneously at 800Ghz. A selected number of nodes have SSD along with HDD.

We setup OrangeFS/PVFS2 [23] to run on 8 nodes of switch 0. The PVFS2 uses HDD of the 8 assigned nodes to stripe the data. In order to mimic the DEP architecture, we choose SDNs amongst available nodes of switch 0, and we choose Compute-side Data Nodes (CDNs) and CNs amongst nodes in switch 1 and 2. We also compare our design to BigSort (Integer sorting in CORAL benchmark [24]). We use the same PFVS2 configuration for BigSort. However, compute nodes for running BigSort is chosen amongst nodes in switch 1 and 2.

TABLE I: System Model Parameters (Result of Micro-Benchmarks)

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

TABLE II: Tunable and Derived Model Parameters

Parameter	Value
NUM_FS	1,2,4
NUM_IO	1,2,4
NUM_CN	4,8,12,16,20,24
CORES_PER_NODE	8
DRAM_ALLOCATION	1 GiB
FS_BUFFER_SIZE	32,64,128 MiB
FS_READ_SIZE	16,32 MiB
CN_MERGE_BUFFER	200 MiB
CHUNK_SIZE	16,20 MiB
CN_MMERGE_BUFFER	eq. to FS_BUFFER_SIZE
TYPE_SIZE	4 bytes (integers)
FILE_SIZE	$2 \times (NUM\_FS + NUM\_IO + NUM\_CN)$ GiB
CN_RECV_BUFFER	eq. to $\frac{FS\_BUFFER\_SIZE}{NUM\_CN}$
NUM_CHUNKS	eq. to $\frac{CN\_MERGE\_BUFFER}{CHUNK\_SIZE}$
NUM_SERIES	eq. to $\frac{FILE\_SIZE}{NUM\_CN \times CN\_MERGE\_BUFFER}$

B. Micro-benchmarking and System Parameters

Here is the list of system parameters and a description of the micro-benchmark we ran to find out about the parameter:

- **Access time to parallel file storage and also to SSD.** This is achieved by measuring the time it takes to write/read a small portion to/from a random location of a file.
- **Bandwidth to parallel file storage and also to SSD.** This is achieved by running ioda.c (given in HW10 in the course) and considering the level-3 collective IO write as the bandwidth to disk. Note that if we run the benchmark on nodes within the same switch of 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 bandwidth seen by the compute nodes. The result of this benchmark even more signifies the benefits of DEP architecture. In general, it is expected that the SDNs get a higher bandwidth to parallel file system due to their proximity.
- **Memory bandwidth.** We ran STREAM to find this out.
- **Compression and decompression bandwidth.** We ran the compression algorithm on 128 MiB of data to find this out.

Table I contains the full list of values of system parameters obtained by running micro-benchmarks.

C. Performance Tuning

There are two angles for performance tuning: tuning for buffer sizes, and tuning for ratio of number of CNs to CDNs and SDNs. We used the performance modeling to tune the size of various buffers we have in different types of nodes. Note that 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 DRAM_ALLOCATION. Table II contains the values of the system parameters, specially buffer sizes that is suggested by the performance model.

Figure 3 shows the effect of number of CDNs and SDNs on the performance. It is clear that the more CDNs or SDNs results in the better performance. In the extreme case where number of CNs, CDNs, and SDNs are all equal, each SDN/CDN is responsible to serve one and only one CN. This causes the minimum amount of contention on CDNs, and maximizes the parallelization. However, in the real 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 possible. Because of this reason, for the rest of our experiments, on the fairly small scale IIT-HEC cluster, we assume that the number of SDNs and also the number of CDNs is 4. Note that the performance model and the actual execution numbers are very close. This is helpful to predict the optimum number of CNs, CDNs, and SDNs on a cluster for each run of our sort algorithm.

D. Scalability Experiments and Comparison with BigSort

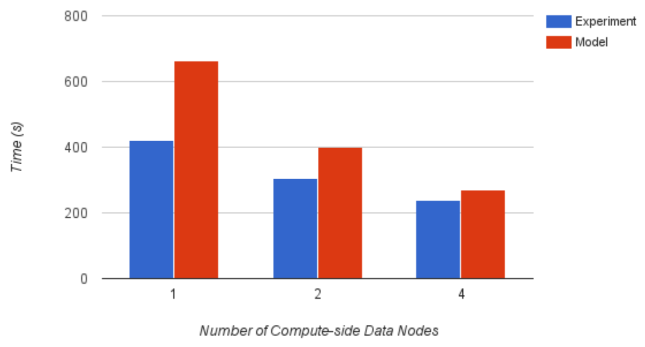
Figure 4 depicts the weak scalability graph for execution time and sorting rate for our implementation and BigSort. Note that the total number of nodes for DEPSort is the sum of number of CNs, CDNs, and SDNs.

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

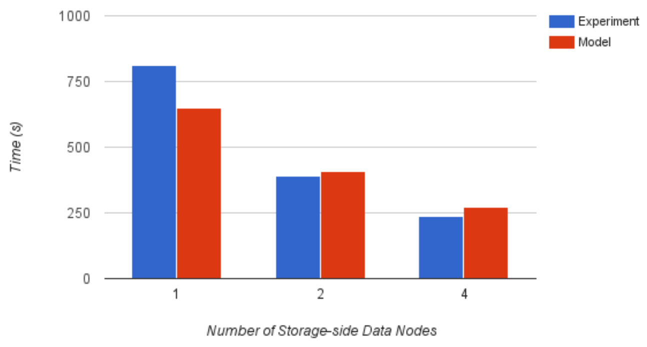
Second, the average rate at which our algorithm sorts the data is 71 MB/s. Considering that the PVFS2 bandwidth is 162 MB/s, and considering the fact that for sorting the data we have to at least read the entire raw date once and write the entire sorted data back to the disk, the best achievable rate is 81 MB/s. This means our soring implementation overlaps almost entire computation and communication with just reading the raw data and writing back the sorted data.

Third, DEPSort is 4.4X faster than BigSort on average. 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 to the parallel file system, and also the actual sorting algorithm reads and writes the input multiple times.

Lastly, performance of DEPSort gradually drops at larger number of nodes. This is mainly due to the contention over CDNs and SDNs. In the largest case, we have 20 CNs, 4 CDNs, and 4 SDNs. That is the ratio of 5 for number of



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



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

Fig. 3: Effect of different number of CDNs and SDNs on performance. File size in all these experiments was 16 GiB.

CNs over CDNs/SDNs. Due to the recourse limitations in our experiments we were not able to run the algorithm on larger scale. However, if we could run on larger scale, we would have been able to find the best ratio of CNs over CDNs/SDNs in which the performance is still acceptable.

VI. Conclusion

In this project we designed and implemented an optimized algorithm for disk-to-disk sorting in the context of DEP architecture. We presented a detailed performance modeling of our algorithm and showed that the performance modeling is a close match with the actual execution time of the algorithm. We compared our algorithm with BigSort (the IntegerSort suite in CORAL benchmark) and showed more than 4X speed-up. Based on our performance results, the time it takes to sort the data in our sorting algorithm is almost equivalent to the time of reading the raw data and writing back the sorted data. That means the sorting algorithm we present in this project, almost entirely overlaps the computation and communication with initial reading from and final writing to the parallel filesystem.

References

[1] 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*, vol. 180, no. 1. IOP Publishing, 2009, p. 012048.

[2] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342010391989

[3] R. Ross, R. Thakur, W. Loewe, and R. Latham, "Parallel i/o in practice," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 216.

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

[5] 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].

[6] 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*. IEEE, 2012, pp. 774–785.

[7] 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*. IEEE Computer Society Press, 2012, p. 35.

[8] 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*. IEEE, 2011, pp. 743–752.

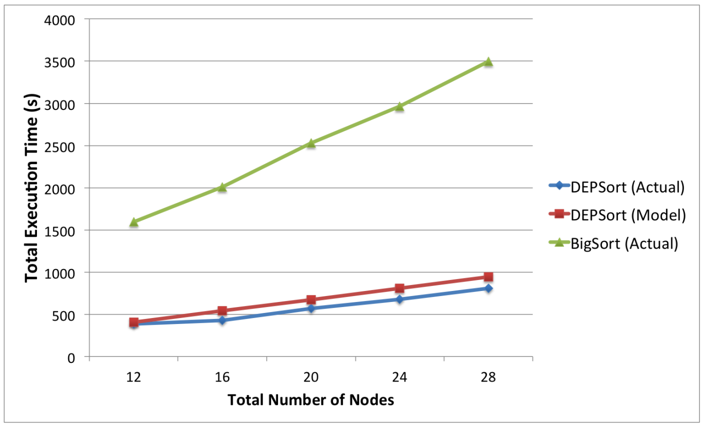
[9] M. W.-u. Rahman, X. Lu, N. S. Islam, R. Rajachandrasekar, and D. K. D. Panda, "Mapreduce over lustre: Can rdma-based approach benefit?" in *Euro-Par 2014 Parallel Processing*. Springer, 2014, pp. 644–655.

[10] J. Sparks, H. Pritchard, and M. Dumler, "The cray framework for hadoop for the cray xc30."

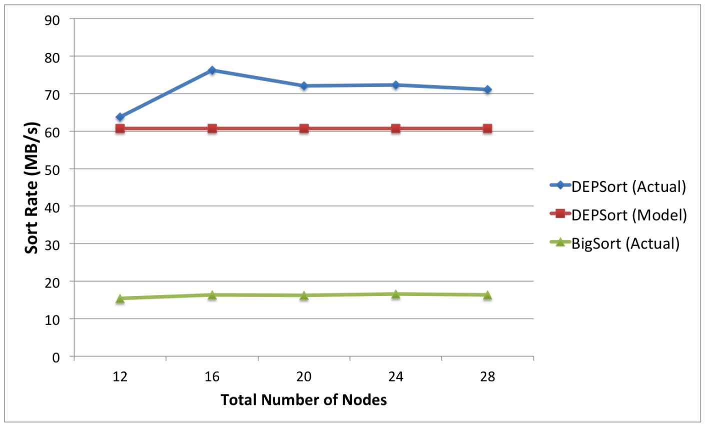
[11] R. Castain and O. Kulkarni, "Mapreduce and lustre: Running hadoop in a high performance computing environment," in *Intel Developers Forum*, vol. 2013, 2013.

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

[13] Y. Xie, K.-K. Muniswamy-Reddy, D. Feng, D. D. Long,



(a) Sort Execution Time



(b) Sort Rate

Fig. 4: Weak scaling result of DEPSort and BigSort. For DEPSort, total number of nodes is equal to the sum of number of CNs, CDNs, and SDNs. Also, for DEPSort experiments, we use 4 CDNs and 4 SDNs. DRAM_ALLOCATION for both DEPSort and BigSort is 1 GiB. For each experiment, the file size is twice the aggregate memory allocation on the entire nodes (i.e. for 12 nodes we use 24 GiB files, for 16 nodes we use 32 GiB files, and so on.)

Y. Kang, Z. Niu, and Z. Tan, “Design and evaluation of oasis: An active storage framework based on t10 osd standard,” in *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*. IEEE, 2011, pp. 1–12.

[14] 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*. IEEE, 2010, pp. 1–12.

[15] B. Rich and D. Thain, “Datalab: transactional data-parallel computing on an active storage cloud,” in *Proceedings of the 17th international symposium on High performance distributed computing*. ACM, 2008, pp. 233–234.

[16] 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*. IEEE, 2010, pp. 1–10.

[17] J. Dongarra *et al.*, “Mpi: A message-passing interface standard (version 3.0),” 2012.

[18] “Partitioned global address space,” http://pgas.org/, accessed: 2014-03-12.

[19] T. El-Ghazawi and L. Smith, “Upc: unified parallel c,” in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 27.

[20] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: An object-oriented approach to non-uniform cluster computing,” *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[21] 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*. IEEE, 2012, pp. 200–208.

[22] 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.

[23] 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*, 2000, pp. 391–430.

[24] “Coral benchmark,” https://asc.llnl.gov/CORAL-benchmarks/, accessed: 2014-03-12.