ExPBB: A framework to explore the performance of Burst Buffer

George S. Markomanolis
Supercomputing Laboratory
King Abdullah University of Science and Technology
Thuwal, Saudi Arabia
georgios.markomanolis@kaust.edu.sa

***Abstract*—ShaheenII supercomputer provides 268 Burst Buffer nodes based on Cray DataWarp technology. Thus, there is an extra layer between the compute nodes and the parallel filesystem by using SSDs. However, this technology is new, and many scientists try to understand and gain the maximum performance. We present an auto-tuning I/O framework called Explore the Performance of Burst Buffer. The purpose of this project is to determine the optimum parameters to acquire the maximum performance of the executed applications on the Burst Buffer. We study the number of the used Burst Buffer nodes, MPI aggregators, striping unit of files, and MPI/OpenMP processes. The framework aggregates I/O performance data from the Darshan tool and MPI I/O statistics provided by Cray MPICH, then it proceeds to the study of the parameters, depending on many criteria, till it concludes to the maximum performance. We report results, where in some cases we achieved speedup up to 4.52 times when we used this framework.**

***Keywords*-DataWarp, Burst Buffer, I/O, performance, optimization**

I. Introduction

Nowadays, the rapid increase in processorsâĂŹ performance, improves the execution of applications significantly but does not happen the same with the memory and storage technology; thus some bottlenecks remain. ShaheenII supercomputer at KAUST includes in its architecture 268 Bust Buffer (BB) nodes based on Cray DataWarp (DW) [1] technology providing a total capacity of 1.52PB. Cray DataWarp applications I/O accelerator adds another layer between the compute nodes and the parallel filesystem by using SSDs disks. In this paper, we present an auto-tuning I/O framework called Explore the Performance of Burst Buffer (ExPBB). We extensively study how to use the Burst Buffer efficiently from many aspects and demonstrate the performance that we achieve with benchmarks/applications. From the results, we also illustrate the need for advanced I/O libraries to achieve high performance on Burst Buffer.

II. Motivation

During the exploration of the DataWarp technology and the effort to achieve the better performance, there were many challenges with the most important, to understand how this technology works. National Energy Research Scientific Computing Center (NERSC) have done significant work through NERSC Burst Buffer Early User Program [2]. They have shown cases where the Burst Buffer achieves better performance than Lustre. Moreover, they have identified various challenges to achieve better performance. We experienced similar challenges.

In the first DataWarp software there a few issues that were solved with the Cray support. However, it was obvious that achieving good performance was not quite straight forward and especially for a user who has no previous experience on I/O optimization. There are plenty of optimization parameters that could be used to improve the I/O performance. A scientist needs to use an appropriate software stack to take advantage of some developments but is also required to understand how they work in order to take decisions for their values. Thus, it was decided that it is important to develop a framework that could propose to the user the optimum values for his application. This way, the Burst Buffer will be used more efficient from the users.

III. Applications

During this work, we use one benchmark and two applications. The first one is called NAS Parallel Benchmarks Block-Tridiagonal (BT) I/O [3], and it presents a block-tridiagonal partitioning pattern on a three-dimensional array across a square number of processes. Each process handles multiple Cartesian subsets of the entire data set, and they increase with the square root of the number of processes participating in the computation. Multiple global arrays are consecutively written to a shared file by appending one after another. The number of global arrays can be adjusted; more information is provided in [4]. We are interested in studying the I/O performance withs Parallel NetCDF (PnetCDF) format. Thus we chose an implementation of BT I/O which employs this format [5]. Weather Research and Forecasting Model (WRF) [6] is one of the most used models in Earth Sciences related fields. WRF code consumes a significant amount of core-hours on many supercomputers, and we study WRF-CHEM which is WRF coupled with chemistry. For this study, we use a WRF-CHEM v3.7.1 and a domain that is a real case of a user. Finally, the demonstration of Parallel version of IDX, called PIDX [7], shows that more advanced I/O libraries are required for better performance. For our experiments, we compile the applications with Cray-MPICH v7.4.2, Cray compiler v8.5.2 for the NAS BT I/O

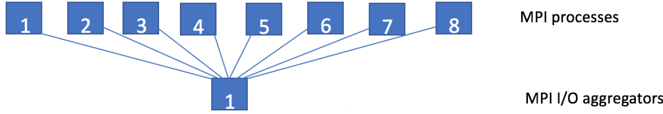

Figure 1. Example of 8 MPI processes writing data on Lustre file system without striping files.

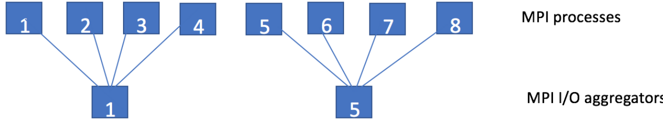

Figure 2. Example of 8 MPI processes writing data on Lustre file system with striping equal to 2.

and PIDX, and Intel v16 for WRF. We are also using and Parallel-NetCDF 1.7.0.

IV. Identifying parameters to be studied

A. *MPI processes*

There are many parameters that can influence an application's performance and especially I/O. Depending on if an application is developed with pure MPI or MPI/OpenMP, we could have more or less MPI processes per compute node. The link from the compute node on Aries network can be influenced by many MPI requests that share links as also if the network is used significantly by other applications etc.

B. *MPI I/O aggregators*

Many parallel applications save data to hard disks by using MPI I/O directly or through libraries such as Parallel NetCDF (PnetCDF) [8], Parallel HDF5 (PHDF5) [9], or ADIOS [10]. It is a common technique that when there is MPI I/O, and it is under collective mode, a part of the MPI processes gather data, and then they save the final shared file. These MPI processes are called MPI I/O aggregators. On Cray supercomputers with Lustre filesystem, the default number of the MPI I/O aggregators is the number of the Lustre stripes that is used for the files. So, if we do not stripe the files, then one MPI process aggregates all the data and saves them to the hard disk as shown in Fig. 1. This is a critical drawback as except that there is communication towards one MPI process only, the I/O occurs only from one MPI process even if a parallel I/O is implemented by all MPI processes.

In the case that the files are striped through the appropriate procedure, then the MPI I/O aggregators are more than one for Lustre, and more than the number of BB nodes for Burst Buffer. In Fig. 2 we use 8 MPI processes, and the file striping is equal to 2. In this case, two MPI processes are aggregating the data and writing the files on hard disks.

The information on the two previous Figures was validated by CrayPAT [11] profiling tool where we could acquire the number of how many MPI processes are writing files on the hard disks. There are two ways to stripe files; either by the command *setstripe* on the folder that I/O takes place, or by using the environment variable *MPICH_MPIIO_HINTS* and declaring which files should be striped. However, on Burst Buffer the filesystem internally is different than Lustre. So, by default, the number of file striping, is equal to the number of reserved BB nodes. This means that if we reserve one BB node, which is constituted by 2 SSD hard disks, then one MPI process saves data, so we cannot achieve high I/O bandwidth. Starting on Cray MPICH 7.4.0, Burst Buffer supports the environment variable *MPICH_MPIIO_HINTS*, so we can declare more MPI I/O aggregators per BB node. In addition to the previous information, with the environment variable shown in Listing 1 we can extract a list of nodes which include MPI I/O aggregators.

```
export MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1
```

Listing 1. Printing in the output all the nodes who are handled as MPI I/O aggregators

Usually, in order to stress further the I/O bandwidth, we need to use at least four MPI I/O aggregators per BB node, for example, the command in Listing 2 declares that all the filenames starting with the word *output*, will be striped over four BB instances. It is important that the number of compute nodes could be divided by the number of MPI I/O aggregators to avoid any I/O imbalance.

```
export MPICH_MPIIO_HINTS=``output*:cb_nodes=4"
```

Listing 2. Declaring the striping of all the output files over 4 DataWarp nodes

In order to have a better load balance of I/O data aggregation, it is necessary that the number of used compute nodes is multiple of the number of the BB nodes. The jobs using BB nodes have to compete with all other jobs for utilizing the bandwidth of the Aries interconnect. The highest number of BB nodes does not always mean a better I/O performance, as the Cray Aries interconnect network and its non-dedicated usage could provide poor results. If the output file is not big enough, then increasing the number of BB nodes, will not deliver the expected performance. In Fig. 3 we show the results when we save one PnetCDF file of 50 GB, by using 256 MPI processes on 8 compute nodes, we use one BB node, and we increase the number of the MPI I/O aggregators. We observe that the default value of the MPI I/O aggregators achieves I/O bandwidth around to 640 MB/s, while for 32 aggregators, we have 2260 MB/s, 3.5 times better performance. Thus, the MPI I/O aggregators are quite important for performance study on Burst Buffer.

C. *Number of Burst Buffer Nodes*

It is already mentioned that ShaheenII is constituted by 268 BB nodes. However, it does not mean that more BB nodes, better the performance. Except, that the performance depends on the Aries interconnection if there are not enough

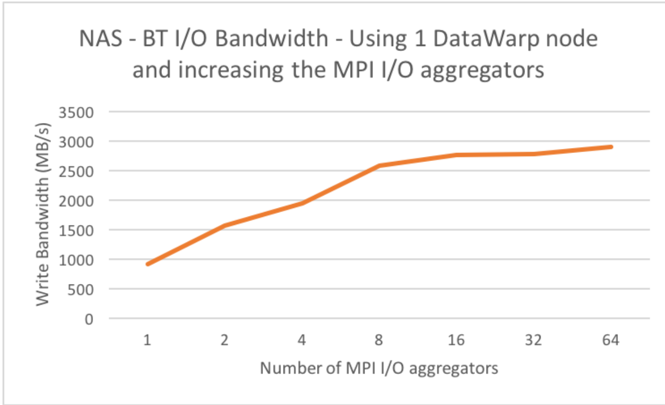

Figure 3. I/O Bandwidth using one DataWarp node and increasing the MPI I/O aggregators.

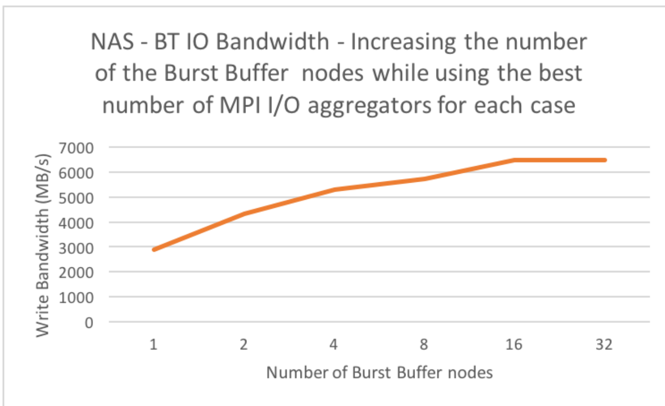

Figure 4. I/O Bandwidth using one Burst Buffer node and increasing the number of BB nodes.

data to be distributed across the BB nodes and if the parameters are not optimized, then we will not achieve the expected performance. For example, in Fig. 4, we increase the number of the Burst Buffer nodes, while we use the best value for the MPI I/O aggregators in each case. We observe that the I/O bandwidth from almost 3000 MB/s becomes around to 6600 MB/s by increasing 32 times the used resources, a result which is not impressive considering the usage of the extra resources.

One significant phase of the ExpBB tool is to investigate what is the most efficient number of BB nodes for an executed application and problem size.

D. Striping Unit

The stripe units are the segments of sequential data written to or read from a disk before the operation continues to the next disk. The amount of the data in one stripe unit is called stripe size. The default value of the stripe size on Lustre is 1 MB and on Burst Buffer is 8 MB. In Fig. 3 an example is presented how to change the striping unit for the files with filename *output** to 4 MB, the value is declared in bytes.

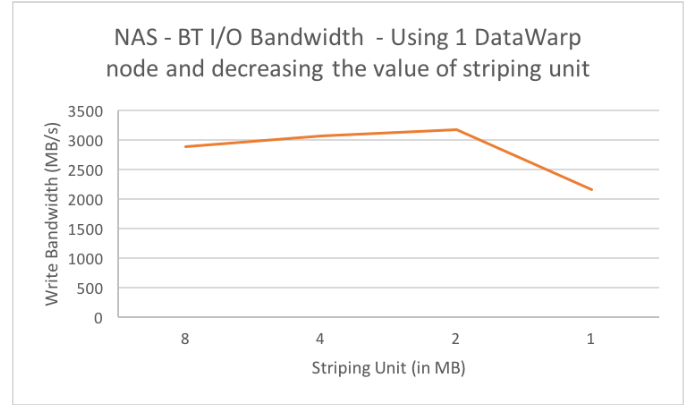

Figure 5. I/O Bandwidth using one BB node and decreasing the striping unit.

```
export MPICH_MPIIO_HINTS="output*:striping_unit=4194304"
```

Listing 3. Declaring the striping unit of all the output files to 4 MB (unit in bytes)

Smaller the striping unit, more write or read operations are required. Depending on the case, this can improve the performance up to one point that the SSDs are stressed, and more resources should be used. In Fig. 5 we show the I/O bandwidth by using one BB node and decreasing the striping unit of the output file.

E. Striping Buffer

During MPI I/O, as we mentioned above, the data are aggregated to the MPI I/O aggregators. When the buffer is full, then the actual I/O operation occurs. Thus, increasing the size of the buffer, more I/O calls will be aggregated and will have less write/read operations. In Fig. 4 an example is presented how to change the striping buffer for the files with a filename starting with *output** to 16 MB, the value is declared in bytes.

```
export MPICH_MPIIO_HINTS="output*:striping_buffer=16777216"
```

Listing 4. Declaring the striping of all the output files over 16 MB of striping buffer

The change of the value of striping buffer does not always help significantly, but on large files across many BB nodes we have seen improvements up to 25% because of doubling the default striping buffer, which is 16MB.

V. Methodology

The ExpBB tool handles all the previous presented parameters in order to conclude to the most efficient combination with regard to the application's performance. This problem is multi-parametric, and the main target is to decrease the I/O time but at the same time to decrease the total execution time. The second one is more important, and we will explain later what does this mean. In order to describe better how the tool works, we'll provide an example step by step of

what data it investigates. We assume that a user of this tool is not experienced with the Burst Buffer.

A. Preparation and Restrictions

The first version of tool, which is still under development, is available in [12]. The current work corresponds to ExpBB v1.0. The user is responsible for two initial steps of preparation before the execution of the tool. Initially, the application should be compiled with CRAY-MPICH 7.4.0 and later, and also the Darshan [13] tool should be activated. Darshan is a profiling tool that provides insights into the I/O performance. It is used from the presented framework for various metrics. The second step is to define into the main script of the framework, called *expbb*, the first parameters. For example, the name of the executable, the required arguments, and the minimum size of BB space in GB which is required to save all the input and output files. The user should define if an investigation of the MPI/OpenMP parameters should take place, the minimum compute nodes necessary for this problem size, and the *stage-in* and *stage-out* paths. The current restriction of the tool is that we can study the I/O for one file per experiment. This is done because the studied parameters can be different per file, especially when they do not have similar size and makes the problem more complicated. For the sake of simplicity, we study more detailed the benchmark NAS BT I/O, because the submission script is smaller. In Listing 5 we present a simple submission script of NAS BT I/O which works for Lustre filesystem where its filename is *btio.sh*.

```
#!/bin/bash
#SBATCH --partition=workq
#SBATCH -t 06
#SBATCH -A k01
#SBATCH  --ntasks=256
#SBATCH --ntasks-per-node=32
#SBATCH   -J btio
#SBATCH -o btio_out_%j
#SBATCH -e btio_err_%j

cp temp_input inputbt1.data
export curr_path=$PWD
echo $curr_path >> inputbt1.data

srun --ntasks=256 --ntasks-per-node=32 \
--threads-per-core=1 --hint=nomultithread \
./btio inputbt1.data
```

Listing 5. Initial submission script for executiong NAS BT I/O on 256 MPI processes.

B. Executing the ExpBB tool

Now the user executes the main script to prepare the new submission scripts

```
./expbb btio.sh
```

Listing 6. Executing the ExpBB tool

During the execution of the ExpBB, we have two main phases. In the first phase, the application is executed two times, one on Lustre and one on Burst Buffer with default values, taken under consideration the minimum requirements from the user. Moreover, for all the scripts that this tool uses, we add automatic the environment variables presented in Listing 7.

```
export MPICH_VERSION_DISPLAY=1
export MPICH_ENV_DISPLAY=1
export MPICH_MPIIO_HINTS_DISPLAY=1
export MPICH_MPIIO_STATS=2
export MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1
```

Listing 7. MPI environment variables that provide extra information relate to the MPI-I/O of the application

The variable *MPICH_ENV_DISPLAY* displays the values of all the adjustable MPI variables that a user can modify. This is useful for reproducible reasons when someone wants to repeat an experiment, and maybe a value was optimized by the user during the execution. The variable *MPICH_MPIIO_HINTS_DISPLAY* prints the I/O hints and their values. So we could identify the default values and know every moment what values we are using. The variable *MPICH_MPIIO_STATS* activates more MPI I/O statistics that we can analyze and are useful to identify I/O issues. The variable *MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY* displays all the MPI I/O aggregators. This helps a user to identify any issue that declared less or more aggregators by mistake. With the first two initial executions and the output files, the tool extracts the default values of the file systems for the environment variables, such as striping unit and buffer. Then, these values are used to explore approaches for better performance. When these executions finish, the framework extracts the available BB nodes from the system (excluding the drained ones). Then it creates new submission scripts, with the minimum required BB nodes, and doubling them till it reaches the maximum number of available BB nodes. For example on ShaheenII, we have 268 BB nodes, if the minimum requested BB nodes is 1, then we'll have submission scripts for 1, 2, 4, 8, 16, 32, 64, 128, 256 BB nodes. For each submission script, the corresponding SLURM DataWarp commands are declared by the tool without the user being familiar with them, the commands before the *srun* call remain, and we add all the new code just before and after the *srun* call. The original code which was after the *srun* command, is added in the end of the script.

After the execution of the command in Listing 6, we have new submission files, for example for 32 BB nodes, is called *expbb_1_32_btio.sh*, where 32 defines the number of the BB nodes, and the number 1 is internal variable of the tool.

The new submission file for the 32 BB nodes looks like in Listing 8. This is a part of the real file which is more than 140 lines. The lines 11-15 include the SLURM DataWarp commands to reserve the appropriate space and declare the *stage-in/out* paths without the user knowing anything about Burst Buffer, except declaring the related paths in the *expbb* script. The lines 17-23 include commands that declare variables. Line 25 is calling a script that a user

is responsible for preparing for the application in the case that is required to execute an executable from Lustre while is using Burst Buffer, this is called hybrid mode and is explained later. From line 28 are declared MPI processes, OpenMP threads, and a loop with MPI I/O aggregators starting with minimum one aggregator per BB node and stressing them till 8 aggregators per BB node (totally 256). This limit is because we have 256 MPI processes, so there is no point to having more MPI I/O aggregators than the MPI processes. Using so many MPI I/O aggregators, in this case, does not seem to be efficient, but we want to capture any case that an application performs better. We have nested loops for all the previously mentioned studied parameters that the range of their values is arranged according to the extracted default values from initial execution and the user's requirements.

```
#!/bin/bash
#SBATCH --partition=workq
#SBATCH -t 06
#SBATCH -A k01
#SBATCH  --ntasks=256
#SBATCH --ntasks-per-node=32
#SBATCH --ntasks-per-socket=16
#SBATCH   -J btio
#SBATCH -o btio_out_%j
#SBATCH -e btio_err_%j
#DW jobdw type=scratch access_mode=striped capacity=12704GiB
#DW stage_in type=directory source=/project/k01/markomg/development/expbb \
destination=$DW_JOB_STRIPED
#DW stage_out type=directory destination=/project/k01/markomg/back2 \
source=$DW_JOB_STRIPED

cd $DW_JOB_STRIPED
chmod +x btio
export folder=${SLURM_SUBMIT_DIR}/experiments
export best_run_id=`cat ${SLURM_SUBMIT_DIR}/best_run_id.txt`
export execu=btio
export exp_id=1
export run_id=1

cp temp_input inputbt1.data
export curr_path=$PWD
echo $curr_path >> inputbt1.data

export MPICH_MPIIO_HINTS="btio.nc"
let expbb_mpi_tasks=256
let expbb_omp_tasks=1
if [ "$expbb_mpi_tasks" -gt "$nodes" ]; then
let sockt=$expbb_mpi_tasks/$((2*$nodes))
fi
for (( expbb_io_aggr=32; expbb_io_aggr<=256; expbb_io_aggr=2*$expbb_io_aggr ));
do
export MPICH_MPIIO_HINTS="$MPICH_MPIIO_HINTS:cb_nodes=$expbb_io_aggr"
...
export START=$(date +%s.%N)
time srun --ntasks=$expbb_mpi_tasks --nodes=$nodes \
--ntasks-per-node=$((2*$sockt)) --cpus-per-task=${expbb_omp_tasks}\
 --threads-per-core=1 --hint=nomultithread ./btio inputbt1.data
END=$(date +%s.%N)
DIFF=$(echo $END - $START | bc)

./parse_darshan.sh $SLURM_JOBID $run_id $folder $DIFF
...

   done
   ...
done
```

Listing 8. Initial submission script for executiong NAS BT I/O on 256 MPI processes and 32 BB nodes.

The values of the environment variables that were mentioned in section IV, are declared to be used during the execution of an application, through also the environment variable *MPICH_MPIIO_HINTS*. There is a general rule, for every execution, we compare the I/O time for the studied file, and the total execution time of the previous execution. In theory, if the main I/O time decreases, then the total execution time will decrease. However, with the current DataWarp software (CLE 5.2), this does not happen always. We present an example with WRF-CHEM application.

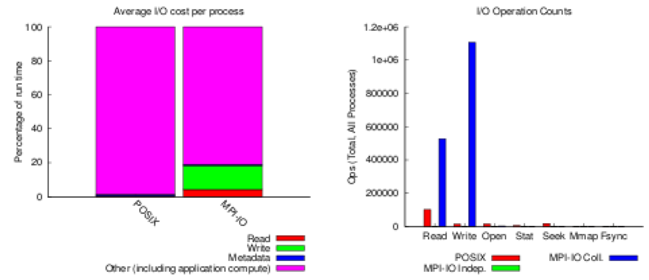

Figure 6. Darshan results on WRF-CHEM with 2 BB nodes.

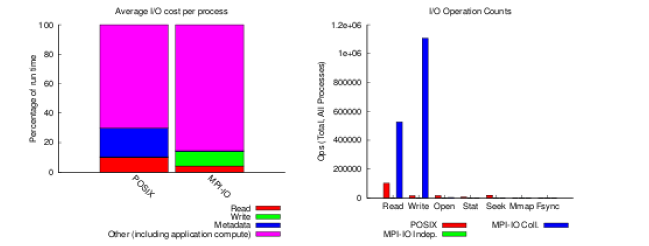

Figure 7. Darshan results on WRF-CHEM with 40 BB nodes.

While the execution time of WRF-CHEM, with 2 BB nodes, is 102 seconds and is presented in Fig. 6, the execution with 16 Lustre OSTs, which is the best result, takes 114 seconds. In Fig. 6, on the left side, is the average I/O cost per process, with the percentage of POSIX and MPI-I/O, while on the right side is the total number of the operations across all the MPI processes and their type.

However, we need to test how this application scales on Burst Buffer. It is quite interesting that while we were trying to test more BB nodes, the missing DVS client of CLE 5.2 caused issues. In Fig. 7 we can see the performance with 40 BB nodes, which is almost 50% worse than 2 BB nodes. This is caused by metadata issues with small size shared access files (blue color on the left side of Fig. 7).

In Fig. 8, we can see the slowest and faster times across the ranks per file and the file size. However, in some cases, the file size is not right. The file namelist.input, which its size is 12KB, needs between 20 and 36 seconds to be read, which is extremely slow. This is caused because of the missing DVS client, which it could be fixed with CLE 6.0 that currently, is not available on ShaheenII.

At this point, we thought to use a hybrid technique, the small files are using Lustre and the large files the BB. However, this approach depends on the application. For WRF-CHEM was possible for most of the files. In Fig. 9, we present the results with 40 BB nodes, that the significant load of metadata of Fig. 7, is not created anymore.

Moreover Fig. 10 shows the range of the slowest and

Variance in Shared Files

File Suffix	Processes	Fastest			Slowest			σ	
		Rank	Time	Bytes	Rank	Time	Bytes	Time	Bytes
.../namelist.input	1280	153	20.415039	22K	0	36.595398	215K	3.81	5.51e+03
..._plev.formatted	1280	1212	0.157083	708	88	9.920437	708	0.839	0
...-04-03_01_00_00	1280	1019	7.685493	0	0	8.316938	19M	0.061	6.3e+06
...-04-03_00_00_00	1280	979	7.503696	0	0	7.511409	19M	0.0554	6.3e+06
...ss/wrfinput_d01	1280	4	5.759557	0	1279	7.327778	0	0.0648	6.06e+06
...e_lat.formatted	1280	1212	0.011889	536	182	6.146939	536	1.86	0
...ozone.formatted	1280	1258	0.013096	531K	580	2.014465	531K	0.609	0

Figure 8. Darshan results on WRF-CHEM per file with 40 BB nodes.

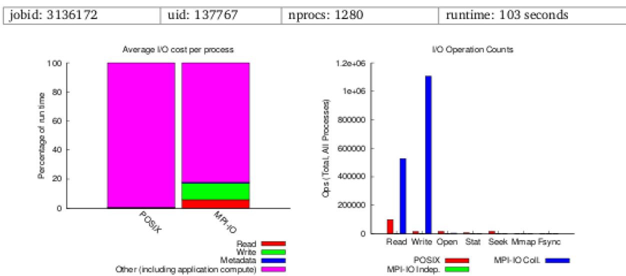

Figure 9. Darshan results on WRF-CHEM in hybrid mode with 40 BB nodes.

fastest ranks across the files where the duration of the small shared files is similar to Lustre and the large files use the Burst Buffer. The total execution time is similar to the usage of the smaller number of BB nodes, but this is normal as the specific problem size is not big enough to take advantage of the extra resources.

From all the previous steps, a new rule was created for the ExpBB framework, when the tool starts testing a larger number of BB nodes, then expect the I/O time of the files, should be checked the total execution time. If there is one file with small size, with shared access, where its I/O time, according to Darshan, is significant, the tool creates a hybrid script, where the application is executed from Lustre, but paths are declared to save large files on Burst Buffer through the *prepare_hybrid.sh* script. However, it is required to be supported by the application, for example, some applications demand that the executable and the files are on the same path. The user should edit the script *prepare_hybrid.sh* and add the necessary commands. Then, the tool it will execute this script when it is necessary. When the hybrid solution is not efficient, the application will be executed again on Burst Buffer mode only.

Variance in Shared Files

File Suffix	Processes	Fastest			Slowest			σ	
		Rank	Time	Bytes	Rank	Time	Bytes	Time	Bytes
...al/wrfinput_d01	1280	1	4.715596	0	1279	7.329664	0	0.1	2.03e+07
...-04-03_01_00_00	1280	1019	5.741487	0	0	6.396622	73M	0.0903	1.32e+07
...-04-03_00_00_00	1280	73	5.923008	0	0	5.972731	73M	0.0873	1.32e+07
.../namelist.input	1280	20	0.000728	22K	1059	0.138662	22K	0.0142	5.53e+03
...ozone.formatted	1280	28	0.040121	531K	1038	0.073565	531K	0.00663	0
...e_lat.formatted	1280	71	0.001312	536	11	0.045462	536	0.00529	0
..._plev.formatted	1280	4	0.001186	708	28	0.045180	708	0.00693	0

Figure 10. Darshan results on WRF-CHEM per file in hybrid mode with 40 BB nodes.

During all the steps the best values are saved in files. Moreover when another number of the BB nodes should be tested, then another job will be submitted which reads from the files the previous optimum results

C. MPI statistics

Cray MPI-I/O provides a way of collecting statistics on the actual read/write performed operations during collective buffering, and it is activated with the environment variable *MPICH_MPIIO_STATS* when is equal to 2. By using this variable, we also create one CSV file per MPI process that can extract in-depth details about an application's I/O with the *cray_mpiio_summary* tool. Listing 9 illustrates the corresponding MPI I/O statistics for writing the file btio.nc on Burst Buffer. We know that the file is located on the Burst Buffer because of the path on the first line. More details are provided, such as the number of independent and collective writes, how many independent writers, the number of the aggregators, the stripe count, and the stripe size. From the variable stripe count, we know how many BB nodes we use for this execution. From the variable aggregators, the number of the MPI I/O aggregators is extracted. In this case is the default, which is one MPI I/O aggregator for each BB node. The stripe size, in this case, is the default one of 8 MB, and usually, we change its value through the *striping_unit* variable. Moreover, two important information are provided by the variables *system writes* and *stripe sized writes*. The first one declares how many write operations occur for this file and the second one how many of them are striped, thus we can extract the efficiency. The ExpBB framework uses the ratio of *stripe sized writes/system writes* to identify the percentage of striped writes (or reads) and conclude to what should be the next step. The variable *total bytes for writes* declares the total amount of the file, while the variable *ave system write size* declares the average system write size and should be close to the stripe size. If it is not, then the tool modifies the stripe size. The number of the gaps and the size is related to how many seek operation happens and how much is the size of the file in bytes that is skipped to read the next part.

```
| MPIIO write access patterns for
|   /var/opt/cray/dws/mounts/batch/3129772/ss//btio.nc
|   independent writes        = 11
|   collective writes         = 40960
|   independent writers       = 1
|   aggregators               = 1
|   stripe count              = 1
|   stripe size               = 8388608
|   system writes             = 6411
|   stripe sized writes       = 6400
|   total bytes for writes
= 53687091532 = 51200 MiB = 50 GiB
|   ave system write size     = 8374214
|   read-modify-write count = 0
|   read-modify-write bytes = 0
|   number of write gaps      = 21
|   ave write gap size        = 23336707978
```

Listing 9. MPI statistics for NAS BT IO

VI. STUDY CASES

A. NAS BT I/O

In this section, we study the performance of NAS BT I/O benchmark on Burst Buffer from user perspective point of view. We will follow a simple methodology which shows how the framework works. We start with a small number of BB nodes, and we will try to increase and observe the performance. However, we will test some optimizations. We know that all the data are not over than 51 GB, so one BB node, is enough. This application reports its I/O write bandwidth for saving 50 GB of data with PnetCDF library, and it is 913 MB/sec with the default settings. In this point, we should mention the practical efficiency of the BB node. By practical efficiency, we mean what performance should we expect by each BB node. We know that IOR benchmark [14] provides around 1.5-1.6 TB/s on 268 BB nodes. In the case, of 1.5 TB/s the practical performance of each node is around to 5.7 GB/s. Just to mention that these results are from an IOR configuration, where its MPI process saves one file, so there is not one single file through collective operations. This means that the practical efficiency of the previous results is around to 15.6% which is quite low. The first step is to investigate if using more MPI I/O aggregators, can provide better performance. We did that already in Fig. 3, and using 64 aggregators provides 2897 MB/s, which means 3.17 times speedup and 50.7% practical efficiency. A user who is not familiar with these technical details, can not achieve this performance. In the continuation, we test the decrease of the striping unit. The second step is again done in Fig. 5 where we decrease the value of the striping unit, and the optimum performance is gained with stripe unit equal to 2 MB and it is 3177 MB/s, almost 10% improvement and the practical efficiency is 54.3%. However, we would like to test why the performance does not increase with striping unit equal to 1 MB in Fig. 5. According to MPI statistics, for striping unit equal to 1 MB, we have the output of Listing 10.

```
| MPIIO write access patterns for
| /var/opt/cray/dws/mounts/batch/3151099/ss//btio.nc
|   independent writes        = 11
|   collective writes         = 40960
|   independent writers       = 1
|   aggregators               = 64
|   stripe count              = 1
|   stripe size               = 1048576
|   system writes             = 51211
|   stripe sized writes       = 51200
|   total bytes for writes
= 53687091532 = 51200 MiB = 50 GiB
|   ave system write size     = 1048350
|   read-modify-write count   = 0
|   read-modify-write bytes   = 0
|   number of write gaps      = 21
|   ave write gap size        = 23297910666
```

Listing 10. MPI statistics for NAS BT IO - 1 BB node - 64 MPI IO aggregators - 1 MB striping unit

We observe that the system writes have been increased from 6411, which were initially, to 51211 for striping unit of 1 MB. In this case, our assumption is that one BB node, can not handle all the load and we should use a second one. By using two BB nodes, the I/O performance for 1 MB striping unit becomes 3850 MB/s, while it was 2165 MB/s for one BB node. However, for striping unit of 2 MB, the I/O write bandwidth with 2 BB nodes, is 4604 MB/s. Thus, the practical efficiency, in this case, is 39.4% and is better to use striping unit equal to 2 MB. We should mention that without the previous optimizations the I/O performance is 1934 MB/s, thus the performance was improved by 2.38 times. All of these procedures occur automatic through the ExpBB framework.

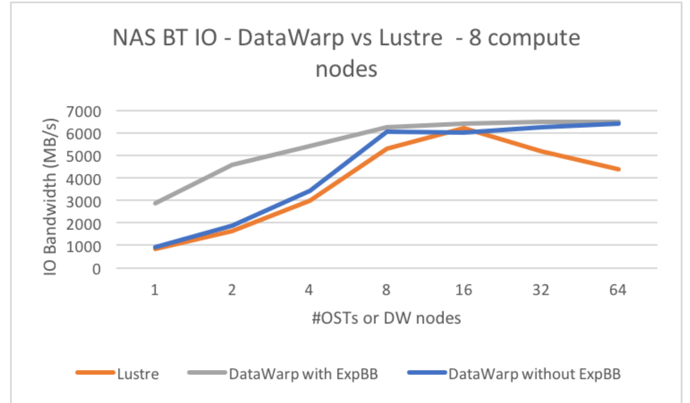

Figure 11. Comparing Lustre vs Burst Buffer and with ExpBB tool for 8 compute nodes

In Fig. 11 we compare the performance of Lustre and Burst Buffer with and without the ExpBB tool. With the framework activated, the performance of Burst Buffer is always better than Lustre from 3.9% for 16 OSTs/BB nodes till 3.39 times for 1 OST/BB node. The comparison of Burst Buffer with and without the tool activated varies from almost similar results till 3.11 times better with first one. From these results, we observe that the default BB settings are quite efficient for 8 BB nodes, where in this case we also use 8 compute nodes, thus we have one MPI I/O aggregator per compute node and corresponds to a dedicated BB node. For this reason, we present more experiments by using same problem size with 32 compute nodes in Fig. 12. In this case, the default settings of BB, work efficient for 32 BB nodes, while we use 32 compute nodes and it seems that this pattern works for this test case. The Burst Buffer with ExpBB is faster than without by 6.2% till 3.28 times. Also the framework provides 1.28 till 4.52 times better performance than Lustre for 64 OSTs/BB nodes and 1 OST/BB node respectively. We did experiments on Lustre for up to 144 OSTs, and it can not achieve so good performance as Burst Buffer for 64 BB nodes.

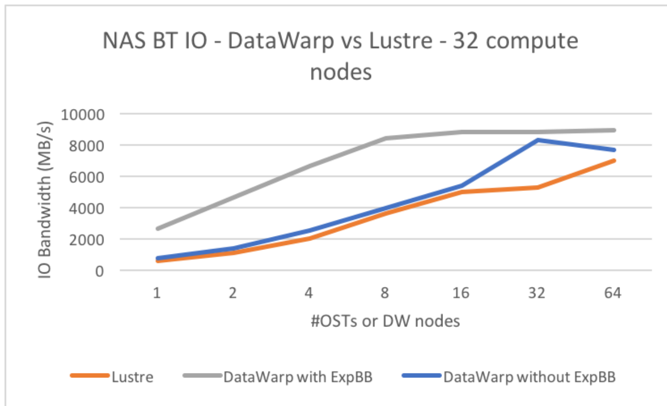

Figure 12. Comparing Lustre vs Burst Buffer and with ExpBB tool for 32 compute nodes

B. WRF-CHEM

This is a real study case which was provided by a user who wanted to test WRF-CHEM on Burst Buffer. However, the domain size is not big enough neither the number of the compute nodes. The user uses 40 compute nodes and produces 2.8 GB output file for one hour of simulation. The total execution time is 164,8 seconds and Listing 11 presents the default statistics by executing this case on 1 BB node where it needs 31,69 seconds to write one output file. The best performance of Lustre takes 112 seconds, so initially, there was no reason to use Burst Buffer for these experiments.

```
| MPIIO write access patterns for
| wrfout_d01_2007-04-03_00_00_00
|   independent writes          = 2
|   collective writes           = 552960
|   independent writers         = 1
|   aggregators                 = 1
|   stripe count                = 1
|   stripe size                 = 8388608
|   system writes               = 797
|   stripe sized writes         = 114
|   total bytes for writes      = 3045341799=2904 MiB=2 GiB
|   ave system write size       = 3821006
|   read-modify-write count     = 0
|   read-modify-write bytes     = 0
|   number of write gaps        = 3
|   ave write gap size          = 8351037

Timing for Writing wrfout_d01_2007-04-03_00_00_00 for
domain          1:    31.69 elapsed seconds
```

Listing 11. MPI statistics for WRF-CHEM on 1 BB node

We can observe that the variable average system write size is a bit less than 4MB, and the percentage of the striped sized writes is only 14,3% . The tool modifies the value of the striping unit to 2 MB and increases the MPI I/O aggregators to 2. Then, we have Listing 12 where the percentage of the striped sized writes is 62,7% and the average system write size is closer to the stripe size. Moreover, now it takes 14,24 seconds to save the file, which means it is 2,22 times faster, and the total execution is 143,8 seconds.

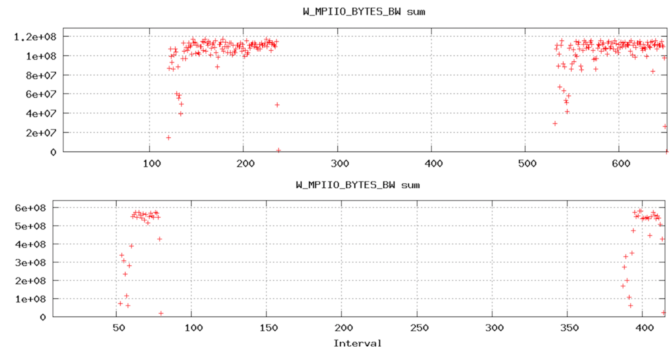

Figure 13. Presenting the MPI I/O bandwidth with time intervals of one simulation hour on Burst Buffer without and with ExpBB

```
| MPIIO write access patterns for
| wrfout_d01_2007-04-03_01_00_00
|   independent writes          = 2
|   collective writes           = 552960
|   independent writers         = 1
|   aggregators                 = 2
|   stripe count                = 1
|   stripe size                 = 2097152
|   system writes               = 1886
|   stripe sized writes         = 1183
|   total bytes for writes      = 3045341799=2904 MiB=2 GiB
|   ave system write size       = 1614709
|   read-modify-write count     = 0
|   read-modify-write bytes     = 0
|   number of write gaps        = 2
|   ave write gap size          = 1048572

Timing for Writing wrfout_d01_2007-04-03_00_00_00 for
domain          1:    14.24 elapsed seconds
```

Listing 12. MPI statistics for WRF-CHEM on 1 BB node

Finally, the tool concludes that the best value for the MPI I/O aggregators is 40 and the striping unit should be equal to 256 KB, with total execution time 97 seconds for using 1 BB node. This result is 13.4% better than the best Lustre execution time with 64 OSTs. The importance of the BB is significant, as we use only one BB node to have better results than Lustre, and for 24 hours of simulation, BB is 14.8% faster than Lustre which means less core-hours consumption. As we have activated the advanced CRAY-MPICH statistics, we use the tool *cray_mpiio_summary*. In Fig. 13 we can observe on the x-axis the time intervals and on the y-axis the write bandwidth for this study case. The BB results without and with ExpBB tool are presented in the upper and lower part respectively. Reading the input file is faster with the ExpBB tool, that's why the writing of the file starts after around to 50 time intervals while they are required more than 100 time intervals without using the tool. Moreover, the write bandwidth is improved 4-5 times. Thus the total execution time is close to 50% faster (less total time intervals).

In Fig. 14 we compare the execution of WRF-CHEM on one node of Burst Buffer without and with ExpBB, using 40 compute nodes, where we have 41.1% performance improvement for the total execution, and around to 73.2%

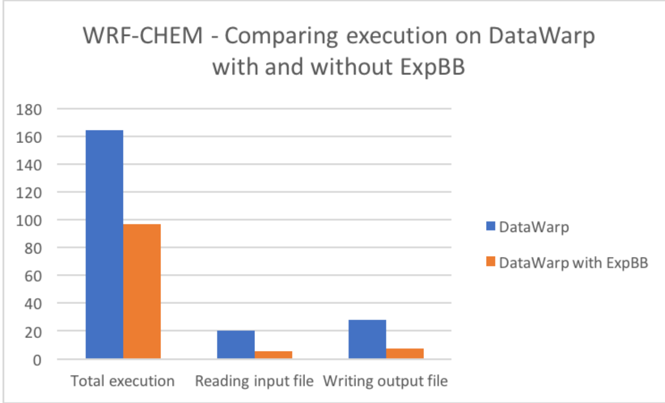

Figure 14. Executing WRF-CHEM on one BB node without and with ExpBB

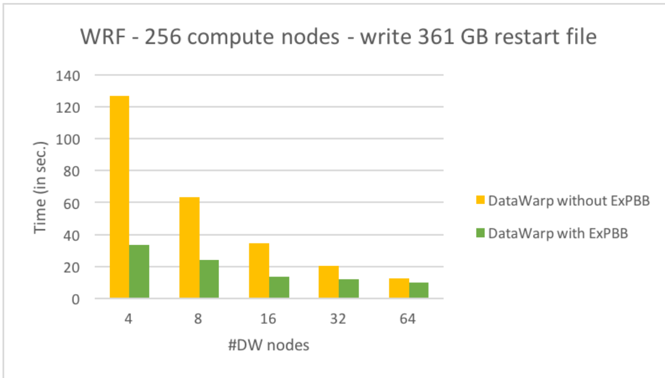

Figure 15. Executing WRF on one BB node without and with ExpBB

for both reading the input and writing the output files.

C. WRF

For this study, we use a benchmark with high-resolution domain - the Alaska domain with 1km resolution [15]. We use WRF v3.7.1 with Cray compiler, MPICH v7.4.2, 256 compute nodes, 4 MPI processes per node and 8 OpenMP threads. WRF can be configured to save the simulation data in history and restart files. The former includes simulation data for specific time steps and the later can be used to restart the simulation. We configure WRF for intensive I/O, so we save the history and restart files every 30 simulation minutes. The size for history and restart files is around 81 GB and 361 GB, respectively. The size of input file (wrfinput_d01) is 77 GB. We stage-in around 110 GB of data, and the stage-out phase after one hour of simulation yields almost 1 TB of data. For the case that we save the data to one single file, we use PnetCDF. In Fig. 15 we study mainly the writing of the restart file, and we present the comparison between DataWarp with default settings and with ExPBB.

The performance is improved 3,8 times on single BB node, and it is 1,28 times faster on 64 BB nodes. As we increase the number of BB nodes, the time percentage of I/O gets smaller and the optimizations gain is smaller. However, the total execution time with 16 BB nodes, is 722 seconds and is faster than 64 BB nodes with default BB settings, which is 737 seconds. So a user can reserve less resources and be charged accordingly. In Listing 13, we present the optimum *MPICH_MPIIO_HINTS* variable for the execution on 64 BB nodes.

```
export MPICH_MPIIO_HINTS="wrfi*:cb_nodes=128:\
striping_unit=4194304, wrfo*:cb_nodes=256:\
striping_unit=4194304, wrfr*:cb_nodes=256:\
striping_unit=4194304"
```

Listing 13. Optimum MPICH_MPIIO_HINTS for WRF on 64 BB node

We observe that some files, need different number of MPI I/O aggregators (cb_nodes) and in some cases will be needed different striping_unit also.

D. Summary

It should be noted that although we present some optimizations analytically, all of these happen automatic with the ExpBB tool. One thing that requests some parts to be done manually is that the tool does not support yet the study of different files on the same execution, so multiple executions should be done per studied file, this will be fixed in a future version. The workflow of the ExpBB tool is presented in Fig. 16. The part of the generated report is not developed yet, but a text file is created with the performance data.

The user executes the ExpBB tool which generates other submission files and then are submitted with dependency mechanism to avoid any overlaps. The framework is tested under CLE 5.2 and Darshan v2.3. For more information check [12]. This framework leads to more efficient execution of an application on Burst Buffer but supports also Lustre. We believe that the users of every level could get advantage of a such auto-tuning tool. This framework integrates sophisticated approaches to stop or advance the search for optimum values of the corresponding parameters.

VII. PIDX

The IDX format provides efficient, cache oblivious, and progressive access to large-scale scientific data by storing the data in a hierarchical Z (HZ) order [16]. In order to study large datasets a parallel version of IDX, called PIDX [7], was developed. To achieve high scalability, the developers of the PIDX framework did implement the total I/O procedure in three phases. Initially, we have the restructuring; blocks of data are created to optimize the layout for I/O. In the continuation, we have the in-core reorganization of data in a read-friendly format following by the data aggregation to optimize disk access. During data restructuring, there is high utilization of the network between the participated processes, and only a subset of them have the required data to participate in next phase. Then, HZ encoding is applied locally on all processes of phase 1. The data aggregation occurs, and the data are written to many IDX files. The

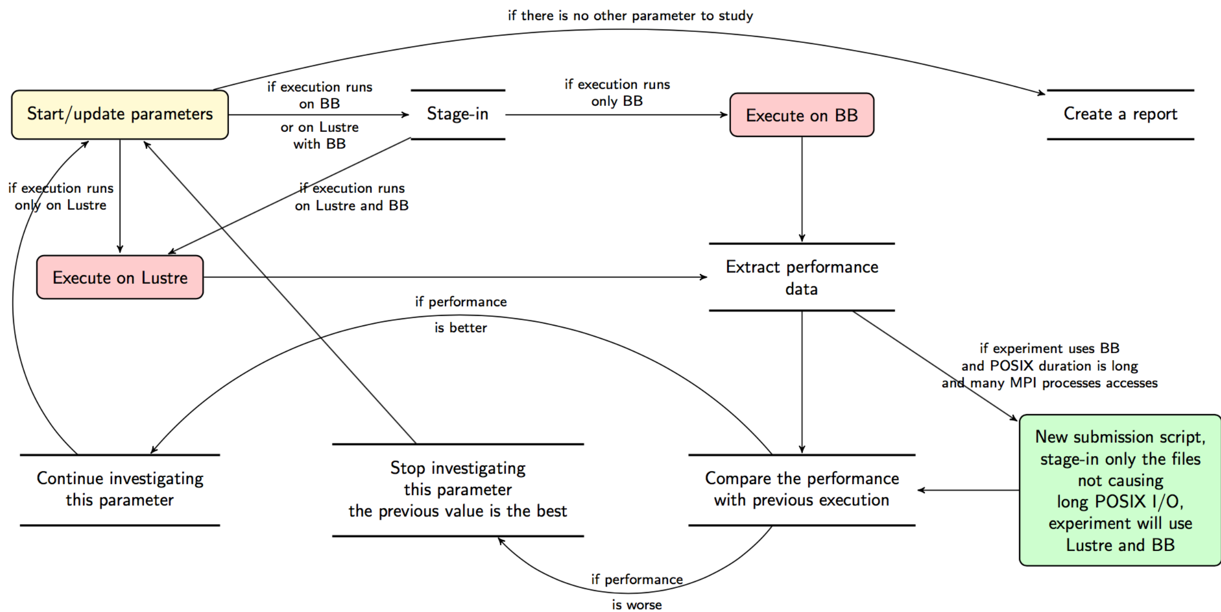

Figure 16. ExpBB workflow

data aggregation is constituted by steps, in which the first one, data are gathered to the I/O aggregators using one-sided MPI communication, and then each aggregator writes its IDX file. This method combines an aggregation strategy that the final phase does not create contention because many files are produced instead of one shared file. We only use a PIDX tutorial that the developers include in the distribution, and they call it *checkpoint_simple*.

This benchmark reproduces the I/O that could be integrated in a real application. In order to produce an average I/O workload per MPI process, that could correspond to a real application, we declare in our experiment that each MPI process handles 64MB of data, so all together the 32 cores, are going to save 2GB of data in a file. Moreover, the 64 MB per process are constituted by 32 variables (2MB for each variable), which illustrate the case that we save in a file the values of 32 variables and each of this one, needs 2MB hard disk space. Fig. 17 presents the results using 144 OSTs on Lustre and 16 to 256 BB nodes.

Moreover, we use 256 compute nodes for 16 BB nodes, till 1024 compute nodes for 256 BB nodes. For 256 compute nodes, we save 512 GB files, while for 1024 compute nodes, we save 2TB files. The requested size of I/O per MPI process remains 64 MB for all the experiments. Although, till 64 BB nodes, the Burst Buffer and Lustre have similar write I/O performance, for more BB nodes, Burst Buffer is scaling while Lustre does not, but it can be a network issue because of other users. More accurate for 256 BB

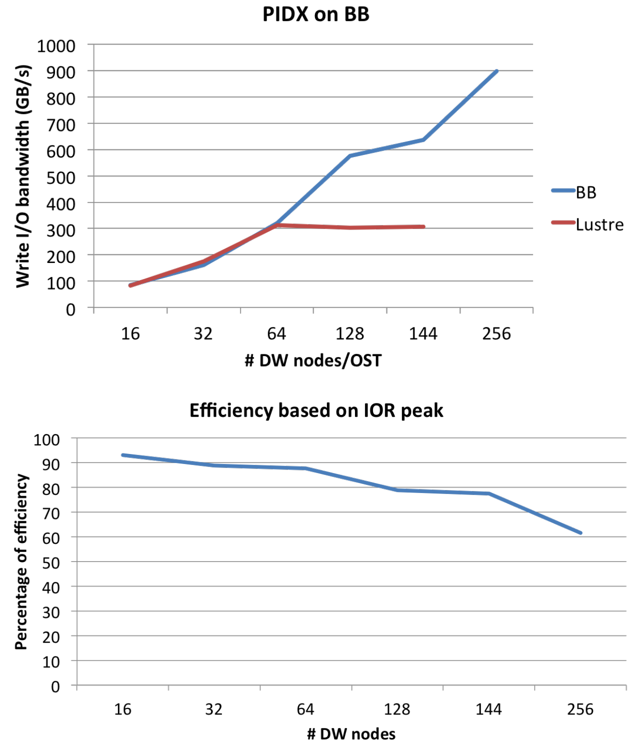

Figure 17. Comparison of Burst Buffer and Lustre for PIDX

nodes, Burst Buffer, achieves 900 GB/s which is three times faster than Lustre's peak, 300 GB/s. All the experiments took place in non dedicated mode, and peak performance can be influenced. The right part of Fig. 17 shows the PIDX I/O efficiency based on IOR peak results. Till 144 BB nodes, the I/O efficiency is above 75%, and it drops a bit more than 60% for 256 BB nodes. As we scale on the system, and with regard to the phase of PIDX that utilizes the network, if the system is busy the results can vary because of the Aries network on XC-40. Thus, we believe that some experiments could be better with dedicated mode and newer compute node Linux which will be available on this system in a few months. From the results, we understand that PIDX is efficient I/O library and it is quite scalable. Thus, it is evident that adapting the I/O approach with other libraries can lead to better results.

VIII. Conclusion

In this work, we present a framework that explores the performance of the Burst Buffer through the study of various I/O parameters. We use parameters that a beginner user maybe is not familiar with them. Thus it can provide support to use Burst Buffer efficiently. This means faster execution time and less core-hours consumption. We know that this tool needs some more development, but it is an ongoing work with promising results. In all the cases using BB with the ExpBB tool lead to better results and in some cases the difference is significant. We do explain some basic functions of BB and which parameters to study as also how to read some performance statistics data. Finally, the study of the NAS BT I/O benchmark and WRF-CHEM model, demonstrated the utilization and the success of the ExpBB tool to improve the performance up to 4.52 times compared to Lustre and 3.28 times compared to Burst Buffer without the activation of the framework.

IX. Future work

Significant future work is planned, although the tool is already operational. It is required to handle more efficient the perfromance study of multiple files. Moreover, the generation of a report is required to have a summary of all the experiments gathered. Some rules that are identified through the study of other applications are going to be integrated. The most significant development will be the integration of some online learning with database. It will be useful if through important criteria we can decrease the range of searching and based to the integrated history of an application to identify the range of some parameters to conclude faster to the optimum solution.

Acknowledgment

For computer time, this research used the resources of the Supercomputing Core Laboratory at King Abdullah University of Science & Technology (KAUST) in Thuwal, Saudi Arabia.

References

[1] Cray, "XC-40, datawarp applications I/O accelerator," http://www.cray.com/sites/default/files/resources/CrayXC40-DataWarp.pdf.

[2] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, S. Byna, S. Farrell, D. Gursoy, C. Daley, V. Beckner, B. V. Straalen, D. Trebotich, C. Tull, G. Weber, N. J. Wright, K. Antypas, and Prabhat, "Accelerating science with the Nersc burst buffer early user program," *Cray User Group*, 2016.

[3] d. W. Wong, P., "R.F.V: NAS parallel benchmarks I/O version 2.4," 2003.

[4] W. keng Liao, "Design and evaluation of MPI file domain partitioning methods under extent-based file locking protocol." *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 2, pp. 260–272, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/tpds/tpds22.htmlLiao11

[5] N. University, "Benchmarking MPI-I/O with PnetCDF on NAS parallel benchmark BT, http://cucis.ece.northwestern.edu/projects/pnetcdf/#benchmarks," 2013, http://cucis.ece.northwestern.edu/projects/PnetCDF/#benchmarks.

[6] J. B. K. J. D. D. O. G. D. M. B. W. W. Skamarock, W. C. and J. G. Powers, "A description of the advanced research wrf version 2," 2005.

[7] S. Kumar, V. Vishwanath, P. H. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. B. Ross, J. Chen, H. Kolla, and R. W. Grout, "PIDX: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets," in *CLUSTER*, 2011.

[8] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 39–. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050189

[9] T. H. Group, "Parallel hierarchical data format," https://support.hdfgroup.org/HDF5/PHDF5/.

[10] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, "Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3125

[11] Cray, "Using cray performance measurement and analysis tools," 2014, http://docs.cray.com/books/S-2376-622/S-2376-622.pdf.

[12] G. S. Markomanolis, "Github: Expbb: A framework to explore the performance of burst buffer, https://kaust-ksl.github.io/expbb/."

[13] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *Trans. Storage*, vol. 7, no. 3, pp. 8:1–8:26, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/2027066.2027068

[14] LLNL, "Ior," $http : //www.csm.ornl.gov/essc/io/IOR-2.10.1.ornl.13/USER_GUIDE.$

[15] D. Morton, O. Nudson, and C. Stephenson, "Benchmarking and evaluation of the weather research and forecasting (WRF) model on the Cray XT5," 2009.

[16] V. Pascucci and R. J. Frank, "Global static indexing for real-time exploration of very large regular grids," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, ser. SC '01. New York, NY, USA: ACM, 2001, pp. 2–2. [Online]. Available: http://doi.acm.org/10.1145/582034.582036