Outline

- Introduction to Parallel I/O
- Understanding the I/O performance on Lustre
- Introduction to Burst Buffer
- Accelerating the performance
## Shaheen II Supercomputer

### Compute

<table>
<thead>
<tr>
<th>Node</th>
<th>Processor type: Intel Haswell</th>
<th>2 CPU sockets per node @2.3GHz 16 processor cores per CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>6174 nodes</td>
<td>197,568 cores</td>
<td>128 GB of memory per node Over 790 TB total memory</td>
</tr>
<tr>
<td>Power</td>
<td>Up to 3.5MW</td>
<td>Water cooled</td>
</tr>
<tr>
<td>Weight/Size</td>
<td>More than 100 metrics tons</td>
<td>36 XC40 Compute cabinets, disk, blowers, management nodes</td>
</tr>
<tr>
<td>Speed</td>
<td>7.2 Peta FLOPS peak performance</td>
<td>5.53 Peta FLOPS sustained LINPACK and ranked 15th in the latest Top500 list</td>
</tr>
<tr>
<td>Network</td>
<td>Cray Aries interconnect with Dragonfly topology</td>
<td>57% of the maximum global bandwidth between the 18 groups of two cabinets</td>
</tr>
</tbody>
</table>

### Storage

<table>
<thead>
<tr>
<th>Storage</th>
<th>Sonexion 2000 Lustre appliance</th>
<th>17.6 Peta Bytes of usable storage Over 500 GB/s bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burst Buffer</td>
<td>DataWarp</td>
<td>Intel Solid State Devices (SSD) fast data cache Over 1.5 TB/s bandwidth</td>
</tr>
<tr>
<td>Archive</td>
<td>Tiered Adaptive Storage (TAS)</td>
<td>Hierarchical storage with 200 TB disk cache and 20 PB of tape storage, using a spectra logic tape library (Upgradable to 100 PB)</td>
</tr>
</tbody>
</table>
Software

- **Application Software**
  - Weather & Environment: WRF, WRF-Chem, HIRAM, MITgcm
  - Big Data: Mizan (in-house)
  - Biology & MD: Amber, Gromacs, LAMMPS, NAMD, VEP, BLAST, Infernal
  - Combustion: NGA, S3D, KARFS
  - CFD & Plasma: Ansys, Fluent, OpenFOAM, Plasmoid (in-house)
  - Chemistry & Materials Science: VASP, Materials Studio, Gaussian, WEIN2k, Quantum Espresso, ADF, CP2K
  - Electromagnetism: Ansys, In-house developed code
  - Oil & Gas: Madagascar, sofi2D, sofi3D, In-house developed codes
  - Seismology: SORD, SeisSol, SPECFEM_3D_GLOBE

- **Development Tool**
  - Compiler: Cray, Intel and GNU with MPICH library
  - Optimized Math Library: Cray-libsci, Intel-MKL, PETSc, FFTW, ParMetis
  - I/O library: HDF5, NetCDF, PNetCDF, ADIOS
  - Performance tools: CrayPat, Reveal, Extrae, Allinea Map
  - Debugger: Totalview, Allinea DDT
Introduction to parallel I/O

- I/O can create bottlenecks
  - I/O components are much slower than the compute parts of a supercomputer
  - If the bandwidth is saturated, larger scale of execution can not improve the I/O performance

- Parallel I/O is needed to
  - Do more science than waiting files to be read/written
  - No waste of resources
  - Not stressing the file system, thus affecting other users
I/O Performance

- There is no one magic solution
- I/O performance depends on the pattern
- Of course a bottleneck can occur from any part of an application
- Increasing computation and decreasing I/O is a good solution but not always possible
Serial I/O

- Only one process performs I/O (default option for WRF)
  - Data Aggregation or Duplication
  - Limited by single I/O process
- Simple solution but does not scale
- Time increases with amount of data and also with number of processes
Parallel I/O: File-per-Process

- All processes read/write their own separate file
  - The number of the files can be limited by file system
  - Significant contention can be observed
Parallel I/O: Shared File

- Shared File
  - One file is accessed from all the processes
  - The performance depends on the data layout
  - Large number of processes can cause contention
Pattern Combinations

- Subset of processes perform I/O
  - **Aggregation** of a group of processes data
  - I/O process may access independent files
  - Group of processes perform parallel I/O to a shared file
Lustre

- Lustre file system is made up of an underlying:
  - Set of I/O servers called Object Storage Servers (OSSs)
  - Disks called Object Storage Targets (OSTs), stores file data (chunk of files). We have 144 OSTs on Shaheen

- The file metadata is controlled by a Metadata Server (MDS) and stored on a Metadata Target (MDT)
Lessons learned from Lustre

- Important factors:
  - Striping
  - Aligned data

- But... how parallel is the I/O?
Collective Buffering – MPI I/O aggregators

- During a collective write, the buffers on the aggregated nodes are buffered through MPI, then these nodes write the data to the I/O servers.
- Example 8 MPI processes, 2 MPI I/O aggregators
How many MPI processes are writing a shared file?

- With CRAY-MPICH, we execute one application with 1216 MPI processes and it provides parallel I/O with Parallel NetCDF and the file’s size is 360GB:

- First case (no stripping):
  - mkdir execution_folder
  - copy necessary files in the folder
  - cd execution_folder
  - run the application
  - Timing for Writing restart for domain 1: 674.26 elapsed seconds

- **Answer:** 1 MPI process
How many MPI processes are writing a shared file?

- With CRAY-MPICH, we execute one application with 1216 MPI processes and it provides parallel I/O with Parallel NetCDF and the file’s size is 360GB:

- Second case:
  - mkdir execution_folder
  - lfs sestripe –c 144 execution_folder
  - copy necessary files in the folder
  - cd execution_folder
  - Run the application
  - Timing for Writing restart for domain 1: 10.35 elapsed seconds

- **Answer: 144 MPI processes**
Extract the list of the MPI I/O aggregators nodes

- export `MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1`

- First case:
  
<table>
<thead>
<tr>
<th>AGG</th>
<th>Rank</th>
<th>nid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>nid04184</td>
</tr>
</tbody>
</table>

- Second case:
  
<table>
<thead>
<tr>
<th>AGG</th>
<th>Rank</th>
<th>nid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>nid00292</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>nid00294</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>143</td>
<td>1144</td>
<td>nid04592</td>
</tr>
</tbody>
</table>
I/O performance on Lustre while increasing OSTs
Declare the number of MPI I/O aggregators

- By default with the current version of Lustre, the number of MPI I/O aggregators is the number of OSTs.

- There are two ways to declare the striping (number of OSTs).
  - Execute the following command on an empty folder
    - `lfs setstripe -c X empty_folder`
      where X is between 2 and 144, depending on the size of the used files.
  - Use the environment variable `MPICH_MPIIO_HINTS` to declare striping per files
    ```
    export MPICH_MPIIO_HINTS="wrfinput*:striping_factor=64,wrfrst*:striping_factor=144,\wrfout*:striping_factor=144"
    ```
Using Darshan tool to visualize I/O performance
Using Darshan tool

- Have you ever used Darshan tool?
  - If the answer is “I don’t know, probably not”, then maybe you have used it, as it is enabled automatic on Shaheen II and Cori.
- KAUST Supercomputing Laboratory (KSL) provides a framework to provide you easy access to performance data from Darshan:
  - Visit web page https://kaust-ksl.github.io/HArshaD/ for instructions. The framework is supported on both Shaheen and Cori, Darshan v2.x and v3.x.
Get the Darshan performance data from your last experiment, execute:
- ./open_darshan.sh

Get the Darshan performance data from the job id 65447, execute:
- ./open_darshan.sh 65447

Compare Darshan performance data from job id 65447 and 65448, execute:
- ./compare_darshan 65447 65448
HArshaD II - Comparison

- In case that you want to compare the execution of two applications, execute:
  - `compare_darshan.sh job1_id job2_id`
  - One PDF file, with the Darshan performance data of both executions, is created.
There are many parameters to optimize Lustre, one quite interesting is the striping_unit. This declares the number of bytes to store on an OST before moving to the next OST.

- `lfs setstripe -s X empty_folder` where X in bytes

- `export MPICH_MPIIO_HINTS="wrfinput*:striping_factor=64,wrfrst*:striping_factor=144:\striping_unit=4194304,wrfout*:striping_factor=144:\striping_unit=2097152"`
Useful MPI environment variables

- `export MPICH_ENV_DISPLAY=1`
  - Displays all settings used by the MPI during execution

- `export MPICH_VERSION_DISPLAY=1`
  - Displays MPI version

- `export MPICH_MPIIO_HINTS_DISPLAY=1`
  - Displays all the available I/O hints and their values

- `export MPICH_MPIIO_AGGREGATOR_PLACEPLACEMENT_DISPLAY=1`
  - Display the ranks that are performing aggregation when using MI-I/O collective buffering

- `export MPICH_MPIIO_STATS=1`
  - Statistics on the actual read/write operations after collective buffering

- `export MPICH_MPIIO_HINTS="…"
  - Declare I/O hints

- `export MPICH_MPIIO_TIMERS=1`
  - Timing statistics for each phase of MPI I/O (requires MPICH v7.5.1)
Burst Buffer

- Shaheen II: 268 Burst Buffer nodes, 536 SSDs, totally 1.52 PB, each node has 2 SSDs
- Adds a layer between the compute nodes and the parallel file system
- Cray DataWarp (DW) I/O is the technology and Burst Buffer is the implementation
Burst Buffer Architecture
Burst Buffer Architecture
Burst Buffer – Use cases

- Periodic burst
- Transfer to PFS between bursts
- I/O improvements
- Accessed via POSIX I/O requests
- Stage-in/stage-out
- Shared BB allocation for multiple jobs
- Coupling applications
Burst Buffer - Status

- **268** DataWarp (DW) nodes, total 1.52PB with granularity 397.44GB

> **dwstat most**
  
  pool units  quantity  free    gran  
  wlm_pool  bytes  **1.52PiB  1.52PiB  368GiB**

  did not find any of [sessions, instances, configurations, registrations, activations]

> **dwstat nodes**

  node  pool online drain  gran capacity insts  activs
  nid00002  wlm_pool  true false  16MiB  **5.82TiB**  0  0

  ...  
  nid07618  wlm_pool  true false  16MiB  5.82TiB  0  0
Check if there are jobs using BB

> scontrol show burst
Name=cray DefaultPool=wlm_pool Granularity=406976M
TotalSpace=1636043520M UsedSpace=0
  Flags=EnablePersistent
  StageInTimeout=1800 StageOutTimeout=1800 ValidateTimeout=5
  OtherTimeout=300
  AllowUsers=...markomg...
  GetSysState=/opt/cray/dw_wlm/default/bin/dw_wlm_cli

Allocated Buffers:
JobID=2729000 CreateTime=2017-01-20T17:15:31 Pool=wlm_pool
Size=813952M State=allocated UserID=markomg(137767)
Per User Buffer Use:
UserID=markomg(137767) Used=813952M

If your username is not in the list of AllowUsers while you have applied for BB early access, send email to help@hpc.kaust.edu.sa
Burst Buffer Nodes Allocation

• How many DW instances per node?

\[
\text{DW\_instances\_per\_node} = \frac{5.82\times 10^24}{368} = 16.19
\]

A DW node can accommodate up to \(16\times 368/1024 = 5.75\ TB\)

• A user requests 60TB of DW nodes, how many DW nodes is he going to reserve (for striped mode explained later)?

We have 268 DW nodes, each nodes provides initially one DW instance and when all of them are used, then it starts from the first DW node again. The allocation occurs under round - robin basis.

\[
\text{Requested\_DW\_nodes} = 60\times 1024/368 = 166.95, \text{ so we will reserve } 167 \text{ DW nodes.}
\]

**Important:** If you reserve more than \(268 \times 368/1024 = 96.31\ TB\), then some DW nodes will be used twice and this can cause I/O performance issues.
Burst Buffer Modes

• DW supports two access modes
  • **Private**
    Each of the compute job has its own private space on BB and it will not be visible to other compute jobs. For now, data is not striped over BB nodes in private mode (not tested). Each compute node has access to a BB allocation equal to the granularity size.
  
  • **Striped**
    The data will be striped over several Burst Buffer nodes. BB nodes are allocated on a round-robin basis. We use this mode mainly

• BB supports two reservation modes
  • **Scratch** is temporary space allocation which will be removed when the job is finished
  • **Persistent** is when you have many jobs that need to access the same files, so this mode creates a DW space that persists after a job is finished and it is available to other of your DW jobs.

*Important*: Persistent space is not a backup solution, you could lose your data in case of any BB problem.
Burst Buffer Workflow

• Initially the files are located on Lustre filesystem

• For the files that need to be accessed multiple times but also for any big files you should move these files on BB before your job reservation. This phase is called **stage-in**. You can stage-in either file or folder.

• When the job finishes, the created files will be returned to the folder that the user declared in the script, this is called **stage-out**.

• The files on BB are located inside the path declared by environment variable `$DW_JOB_STRIPED` (for striped mode)

Note: Stage-in and –out are not mandatory it depends what the user needs. Maybe there are no input files or the user wants just to measure the execution time.
Modify SLURM script

- Lustre reservation

```bash
#!/bin/bash
#SBATCH --partition=workq
#SBATCH -t 10:00:00
#SBATCH -A k01
#SBATCH --nodes=32
#SBATCH --ntasks=1024
#SBATCH -J slurm_test
```

Comment: Insert the DW commands, exactly after the SBATCH commands, do not include any other unrelated commands between SBATCH and DW declarations.
Modify SLURM script

- BB reservation (2TB of DW space)
  ```bash
  #!/bin/bash
  #SBATCH --partition=workq
  #SBATCH -t 10:00:00
  #SBATCH -A k01
  #SBATCH --nodes=32
  #SBATCH --ntasks=1024
  #SBATCH -J slurm_test
  #DW jobdw type=scratch access_mode=striped capacity=2TiB
  #DW stage_in type=directory source=/scratch/markomg/for_bb
destination=$DW_JOB_STRIPED
  #DW stage_out type=directory destination=/scratch/markomg/back_up
source=$DW_JOB_STRIPED/
  
cd $DW_JOB_STRIPED
  chmod +x executable
  ```

Note: You can stage-in/out also files instead of directory
How fast is stage-in?

Cray DataWarp - Stage-in 100GB

- Lustre stripe 1
- Lustre stripe 2
- Lustre stripe 4
- Lustre stripe 8
DataWarp – Restrictions

- When you stage-in executables, you need to execute a command when you are on BB, that this file is executable (`chmod +x executable`)

- Symbolic links will be lost during stage-in
Profiling MPI I/O on BB

**Question:** Using 160 nodes with 1 MPI process per node and 2TB of DW space (6 DW nodes) with MPI I/O through PnetCDF, how many MPI I/O aggregators are saving the NetCDF file on Lustre?

**Answer:** 6!

Table 6: File Output Stats by Filename

<table>
<thead>
<tr>
<th>Write Time</th>
<th>Write MBytes</th>
<th>Write Rate</th>
<th>Writes</th>
<th>Bytes/ Call</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MBytes/sec</td>
<td>MBytes/sec</td>
<td></td>
<td></td>
<td>PE</td>
</tr>
<tr>
<td>710.752322</td>
<td>988,990.668619</td>
<td>1,391.470191</td>
<td>671,160.0</td>
<td>1,545,133.62</td>
<td>Total</td>
</tr>
<tr>
<td>263.824253</td>
<td>369,720.282763</td>
<td>1,401.388533</td>
<td>46,690.0</td>
<td>8,303,272.98</td>
<td>wrfrst_d01_2009-12-18_00_30_00</td>
</tr>
<tr>
<td>45.299442</td>
<td>61,624.000000</td>
<td>1,360.369943</td>
<td>7,798.0</td>
<td>8,286,412.85</td>
<td>pe.96</td>
</tr>
<tr>
<td>44.410365</td>
<td>61,616.000000</td>
<td>1,387.423860</td>
<td>7,795.0</td>
<td>8,288,525.83</td>
<td>pe.160</td>
</tr>
<tr>
<td>43.762797</td>
<td>61,623.999999</td>
<td>1,408.136675</td>
<td>7,763.0</td>
<td>8,323,772.69</td>
<td>pe.32</td>
</tr>
<tr>
<td>43.708663</td>
<td>61,616.148647</td>
<td>1,409.701068</td>
<td>7,762.0</td>
<td>8,323,784.42</td>
<td>pe.0</td>
</tr>
<tr>
<td>43.532686</td>
<td>61,616.134117</td>
<td>1,415.399323</td>
<td>7,764.0</td>
<td>8,321,638.26</td>
<td>pe.128</td>
</tr>
<tr>
<td>43.110299</td>
<td>61,624.000000</td>
<td>1,429.449598</td>
<td>7,808.0</td>
<td>8,275,800.13</td>
<td>pe.64</td>
</tr>
<tr>
<td>0.000000</td>
<td>0.000000</td>
<td>--</td>
<td>0.0</td>
<td>--</td>
<td>pe.1</td>
</tr>
<tr>
<td>0.000000</td>
<td>0.000000</td>
<td>--</td>
<td>0.0</td>
<td>--</td>
<td>pe.2</td>
</tr>
<tr>
<td>0.000000</td>
<td>0.000000</td>
<td>--</td>
<td>0.0</td>
<td>--</td>
<td>pe.3</td>
</tr>
<tr>
<td>0.000000</td>
<td>0.000000</td>
<td>--</td>
<td>0.0</td>
<td>--</td>
<td>pe.4</td>
</tr>
<tr>
<td>0.000000</td>
<td>0.000000</td>
<td>--</td>
<td>0.0</td>
<td>--</td>
<td>pe.5</td>
</tr>
<tr>
<td>0.000000</td>
<td>0.000000</td>
<td>--</td>
<td>0.0</td>
<td>--</td>
<td>pe.6</td>
</tr>
<tr>
<td>0.000000</td>
<td>0.000000</td>
<td>--</td>
<td>0.0</td>
<td>--</td>
<td>pe.7</td>
</tr>
</tbody>
</table>
How do we choose the number of MPI I/O aggregators on BB?

• In this example we have parallel I/O and we can adjust the number of the MPI processes for simulating an application

• MPICH_MPIIO_HINTS
  • export MPICH_MPIIO_HINTS="wrfrst*:cb_nodes=80,wrfout*:cb_nodes=40"

• In this case we select 80 MPI I/O aggregators for the files starting with the name wrfrst*, and 40 MPI I/O aggregators for the files starting with the name wrfout*.

• Although this depends on the application, according to out experience, if you have one MPI I/O aggregator per DW node (default behavior), the performance is not always good. In order to stress the SSDs of the DW node, more than one MPI process should write data per DW node, and this happens with MPI I/O aggregators.

• Depending on the size of the file, some times we need to use different number of MPI I/O aggregators per file.
How do we choose the number of MPI I/O aggregators on BB?

Tips:

- The number of the MPI I/O aggregators should divide the number of total MPI processes for better load balancing. For example, if you have 1024 MPI processes, do not declare 100 MPI I/O aggregators, but 128 or 64.
- The number of the requested DW nodes, should divide the number of the MPI I/O aggregators for better load balancing also.
- Of course the requested DW nodes should provide enough data for all of your experiments, thus there is a minimum amount of needed DW nodes.

\[
\text{MPI\_IO\_aggregators} = \begin{cases} 
\text{DW\_nodes, if we use one aggregator per DW node} \\
 k \times \text{DW\_nodes, where } k \in \mathbb{N}, 2 \leq k \leq 128 
\end{cases}
\]

\[
\text{Number\_of\_total\_MPI\_processes} = l \times \text{MPI\_IO\_aggregators, where } l \in \mathbb{N}, l \geq 2
\]
Compute the required DW space

- It is already mentioned that we need to have enough space for our experiments.
- If the experiments are about DW scalability and the number of the MPI/OpenMP processes remain stable, then you could modify the MPI I/O aggregators and the number of DW nodes. As these two numbers should be divided you can compute how many nodes you have to request.
- If you need for example 64 DW nodes, then you should calculate the requested space as follows:
  - Multiply with the DW granularity:
    - $64 \times 368 = 23552$
Create persistent DW space

- Create persistent DW space

```bash
#!/bin/bash -x
#SBATCH --partition=workq
#SBATCH -t 1
#SBATCH -A k01
#SBATCH --nodes=1
#SBATCH -J create_persistent_space
#BB create_persistent name=george_test capacity=600G access=striped type=scratch
exit 0
```
Checking the status of the persistent DW reservation

> dwstat most

<table>
<thead>
<tr>
<th>sess</th>
<th>state</th>
<th>token</th>
<th>creator</th>
<th>owner</th>
<th>created</th>
<th>expiration</th>
<th>nodes</th>
<th>created</th>
<th>expiration</th>
<th>intact</th>
<th>label</th>
<th>public</th>
<th>confs</th>
</tr>
</thead>
<tbody>
<tr>
<td>985</td>
<td>CA---</td>
<td>george_test</td>
<td>CLI 137767</td>
<td>2017-01-20T18:01:00</td>
<td>never</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inst</th>
<th>state</th>
<th>sess</th>
<th>bytes</th>
<th>nodes</th>
<th>created</th>
<th>expiration</th>
<th>intact</th>
<th>label</th>
<th>public</th>
<th>confs</th>
</tr>
</thead>
<tbody>
<tr>
<td>977</td>
<td>CA---</td>
<td>985</td>
<td>736GiB</td>
<td>2</td>
<td>2017-01-20T18:01:01</td>
<td>never</td>
<td>true</td>
<td>george_test</td>
<td>true</td>
<td>1</td>
</tr>
</tbody>
</table>

> dwstat nodes

<table>
<thead>
<tr>
<th>node</th>
<th>pool</th>
<th>online</th>
<th>drain</th>
<th>gran</th>
<th>capacity</th>
<th>insts</th>
<th>activs</th>
</tr>
</thead>
<tbody>
<tr>
<td>nid01349</td>
<td>wlm_pool</td>
<td>true</td>
<td>false</td>
<td>16MiB</td>
<td>5.82TiB</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>nid01410</td>
<td>wlm_pool</td>
<td>true</td>
<td>false</td>
<td>16MiB</td>
<td>5.82TiB</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
#!/bin/bash
#SBATCH --partition=workq
#SBATCH -t 10
#SBATCH -A k01
#SBATCH --nodes=1
#DW persistentdw name=george_test
#DW stage_in type=directory source=/project/k01/markomg/wrf
destination=$DW_PERSISTENT_STRIPED_george_test

cd $DW_PERSISTENT_STRIPED_STRIPED_george_test/

... 

exit 0
Use DW persistent space II

- Now, you can execute the second job on the persistent space, however, do **not** stage-in the same files:
  - `squeue -u markomg`

<table>
<thead>
<tr>
<th>JOBID</th>
<th>USER ACCOUNT</th>
<th>NAME</th>
<th>ST</th>
<th>REASON</th>
<th>START_TIME</th>
<th>TIME</th>
<th>TIME_LEFT</th>
<th>NODES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2729358</td>
<td>markomg</td>
<td>k01</td>
<td>test</td>
<td>burst_buf</td>
<td>N/A</td>
<td>0:00</td>
<td>3:00</td>
<td>40</td>
</tr>
</tbody>
</table>

- `scontrol show job 2729358`

  ...  
  JobState=PENDING  
  Reason=burst_buffer/cray:_dws_data_in:_Error_creating_staging_object_for_file_(_/scratch/markomg/burst_buffer_early_access/wrfchem/wrfchem-3.7.1_burst/test/em_real/forburst)_-_2_Staging_failures_reported_Dependency=(null)...

- `scancel 2729358`

If the problem is not solved send us email immediately! [help@hpc.kaust.edu.sa](mailto:help@hpc.kaust.edu.sa), inform also the BB users through bb_users@hpc.lists.kaust.edu.sa
Use DW persistent space III

- Submit another jobs by either stage-in different files, or without stage-in

- In the case that you want to connect interactively on the compute node to have access to BB and check the files, follow the instructions:
  - Create a file, called it for example bbf.conf with the following:
    - `#DW persistentdw name=george_test`
  - Execute:
    - `salloc -N 1 -t 00:10:00 --bbf="bbf.conf"`
    - `srun -N 1 bash -l`
    - `cd $DW_PERSISTENT_STRIPED_george_test`
  - `markomg@nid00024:/var/opt/cray/dws/mounts/batch/george_test/ss`
Use DW persistent space IV

- Three jobs were executed on persistent DW space and created a job folder with the job id as their name:
  nid00024:/var/opt/cray/dws-mounts/batch/george_test/ss/ ls -l 2729*

2729356:
-rw-r--r-- 1 markomg g-markomg 3053617664 Jan 20 18:49 wrfout_d01_2007-04-03_00_00_00
-rw-r--r-- 1 markomg g-markomg 3053617664 Jan 20 18:49 wrfout_d01_2007-04-03_01_00_00

2729361:
-rw-r--r-- 1 markomg g-markomg 3053617664 Jan 20 18:57 wrfout_d01_2007-04-03_00_00_00
-rw-r--r-- 1 markomg g-markomg 3053617664 Jan 20 18:57 wrfout_d01_2007-04-03_01_00_00

2729362:
-rw-r--r-- 1 markomg g-markomg 3053617664 Jan 20 19:09 wrfout_d01_2007-04-03_00_00_00
-rw-r--r-- 1 markomg g-markomg 3053617664 Jan 20 19:09 wrfout_d01_2007-04-03_01_00_00
Finalize DW persistent reservation

- When the experiments are finished, then stage-out the files. Do not copy the files from the interactive mode back to Lustre as this can be much slower, depending on the file sizes.

- Finally delete the DW space
Use DataWarp for Metadata intensive jobs

- Real case, a user was hurting the metadata server with just one compute node, reading/writing into the same file more than 140 million times.

- Login nodes almost could not be used, lagging for seconds. Users were reporting slow IO.

- Moving the user to DataWarp, we were able to have many parallel executions of the same job without influencing login nodes or other jobs.
Applications/Benchmarks
Data Centric Optimizations of Seismic Natural Migration Algorithm at Scale on Parallel File Systems and Burst Buffer
Outline

- Seismic Natural Migration
- I/O optimizations
  - on parallel filesystem
  - using Cray DataWarp Burst Buffer
- Summary and Future Work
Seismic Natural Migration

- Natural Migration is a seismic imaging tool that maps buried faults.

- Application to Long Beach, CA area.

Buried fault lines in the subsurface are shown as lineaments in the images. Unknown faults are under populated LA areas.
Seismic Natural Migration

- Natural Migration is a seismic imaging tool that maps buried faults.
- The Algorithm uses recorded Green’s functions $G(s,x,t)$ to compute an image:
  \[\text{Image}(x) = \sum_s^N \sum_r^N [G(s,x,t) \ast G(r,x,t)] \cdot G_1(s,r,t) ,\]
  where the $s$ and $r$ denote seismic data coordinates, and $x$ denotes image coordinates.
- The Green’s functions are pre-computed and stored in a single file with more than 86GB of size (for this experiment)
Computational Aspects

- Natural migration equation: \( \text{Image}(x) = \sum_s^{N} \sum_r^{N} [G(s, x, t) \ast G(r, x, t)] \cdot G_1(s, r, t) \),
- There are \( N=5297 \) Green’s functions.
- The outer summation is distributed among MPI processes.
- All runs are configured with one MPI process per socket using 16 OpenMP threads.
- Each MPI process loads the whole 86GB file in parts (one Green’s function at a time) to compute the inner summation.
- The time convolution and dot-product operations in the equation above are computationally cheap compared to the I/O cost for retrieving the Green’s function from disks.
Natural Migration I/O Profile Before Tuning

![Average I/O cost per process chart]

<table>
<thead>
<tr>
<th>Most Common Access Sizes</th>
<th>File Count Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>access size</td>
<td>count</td>
</tr>
<tr>
<td>17310596</td>
<td>28058209</td>
</tr>
<tr>
<td>4935</td>
<td>58267</td>
</tr>
<tr>
<td>21188</td>
<td>26485</td>
</tr>
<tr>
<td>3268</td>
<td>5297</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tuning Lustre Stripe Count for Natural Migration

![Graph showing execution time vs LFS stripe count for different number of nodes.](image)
Natural Migration I/O Profile After Tuning

### Most Common Access Sizes

<table>
<thead>
<tr>
<th>access size</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>17310596</td>
<td>28058209</td>
</tr>
<tr>
<td>4935</td>
<td>58267</td>
</tr>
<tr>
<td>21188</td>
<td>26485</td>
</tr>
<tr>
<td>3268</td>
<td>5297</td>
</tr>
</tbody>
</table>

### File Count Summary

<table>
<thead>
<tr>
<th>type</th>
<th>number of files</th>
<th>avg. size</th>
<th>max size</th>
</tr>
</thead>
<tbody>
<tr>
<td>total opened</td>
<td>5305</td>
<td>17M</td>
<td>86G</td>
</tr>
<tr>
<td>read-only files</td>
<td>6</td>
<td>15G</td>
<td>86G</td>
</tr>
<tr>
<td>write-only files</td>
<td>1</td>
<td>108M</td>
<td>108M</td>
</tr>
<tr>
<td>read/write files</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>created files</td>
<td>1</td>
<td>108M</td>
<td>108M</td>
</tr>
</tbody>
</table>
Tuning DataWarp Nodes Count for Natural Migration

![Graph showing execution time vs. number of DataWarp nodes for different node counts: 50, 100, 250, 500, and 1000 nodes. The graph indicates a decrease in execution time as the number of nodes increases, reaching a minimum at around 100 nodes for most node counts.](image-url)
Lustre Filesystem vs DataWarp
Summary and Future Work on Seismic Natural Migration Algorithm

- Tuning Lustre stripe count significantly improved the seismic natural migration code, especially at larger scale.
- Natural migration code benefited from DataWarp burst buffer up to a certain scale with up to 34% improvement.
- Next Steps: study the performance of algorithmic changes to minimize I/O and use MPI communications instead.
Study-case NAS BTIO
Applications

NAS BTIO

“As part of the NAS parallel benchmark set an IO benchmark has been
developed which is based on one of the computational kernels. The BT
benchmark is based on a CFD code that uses an implicit algorithm to
solve the 3D compressible Navier-Stokes equations.”
Increasing the file size from 50 GB (2 DW nodes) up to 52 TB (256 DW nodes)
NAS – BT I/O Benchmark - PNetCDF

I/O Efficiency

I/O efficiency (in %)

# DW nodes

I/O Efficiency

66
Applications

- NAS BT I/O
- Domain size: 1024 x 512 x 256
- 256 to 1024 MPI processes, 8 – 32 nodes
- Size of output file: 50 GB
A user tries to scale his application on Burst Buffer by just increasing the BB nodes and this does not always provide the best results.

Increasing the BB nodes by 64 times, provide less than 8 times better performance and the practical efficiency is less than 2%!
Collective Buffering – MPI I/O aggregators II

- Using optimized MPI I/O aggregators improved the performance up to 3.11 times on just one BB node.
- We achieved best performance with 64 MPI I/O aggregators

Use 64 MPI I/O aggregators for the file btio.nc: export MPICH_MPIIO_HINTS=btio.nc:cb_nodes=64
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.

- For NAS BT IO, decreasing the striping unit up to 2 MB, increases the performance by 10%.

Change striping unit of file btio.nc to 2MB:
export MPICH_MPIIO_HINTS="btio.nc:cb_nodes=64:striping_unit=2097152"
NAS BT I/O - Understanding striping unit

2897 MB/s

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        = 64
  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

2165 MB/s

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

By decreasing the striping unit by 8 times, the system writes were increased by 8 times.

Doubling the number of BB nodes from 1 to 2, the I/O bandwidth from 2165 MB/s becomes 3850 MB/s, 78.2% improvement.
CLE comparison

- Cray provides the CLE 6 with new functionalities and performance improvements.
- In the next slides we compare the CLE 5.2 vs 6.0.4
- We use NAS BTIO, with a domain which leads to a shared output file of 25GB.
- We use 1 BB node
With default settings, there is no significant performance difference between the CLE 5.2 and 6.0.4 in this specific case.
I/O Efficiency – Default parameters

I/O Efficiency - NAS BTIO Benchmark - Default parameters
Comparison CLE 5.2 vs CLE 6.0.4 16 MPI I/O aggregators
Shared file - 25GB

MPI processes

I/O Efficiency (in %)

- Write - CLE 5.2
- Read - CLE 5.2
- Write - CLE 6.0.4
- Read - CLE 6.0.4
CLE 5.2 vs 6.0.4 – optimized parameters

By using more MPI I/O aggregators, CLE 6.0.4 achieves up to 3 times better write speed. The performance of reading a file seems similar between the CLE.
I/O Efficiency – Optimized parameters

I/O Efficiency - NAS BTIO Benchmark - Optimized parameters
Comparison CLE 5.2 vs CLE 6.0.4 16 MPI I/O aggregators
Shared file - 25GB
Study-case Neuromap
Application provided for the SC17 tutorial
The Neuronm(ini)app(lication) library reproduces the algorithms of the main software of the Blue Brain Project as a collection of mini-apps. For its first release, the Neuromapp framework focuses on CoreNeuron application.

Replib is a miniapp that mimics the behavior of Neuron's ReportingLib. It uses MPI I/O collective calls to write a fake report to a shared file. The miniapp provides several options to distribute data across ranks in different ways.

Contact person: Judit Planas
The neuromapp application was one of the test codes for an SC17 Tutorial entitled: *Getting Started with the Burst Buffer: Using DataWarp Technology*. The presenters will be George S. Markomanolis from KAUST and Deborah Bard from LBNL.

“The long-awaited Burst Buffer technology is now being deployed on major supercomputing systems. In this tutorial, we will introduce the Burst Buffers deployed at the two latest supercomputers at NERSC (Cori) and KAUST (Shaheen II) based on the Cray DataWarp, and discuss in detail our experience with Burst Buffers from both a system and a user’s perspective. Both KAUST and NERSC have been supporting BB projects for more than a year, and have developed a wealth of experience using these resources efficiently. For this tutorial, we combine the knowledge and experience of staff from both sites to provide attendees with an effective understanding of how to optimally use BB technology. We focus on optimizing massively parallel I/O for SSDs, a relatively new problem compared to well-established optimizations for parallel I/O to disk-based file systems. The tutorial will conclude with a live demonstration of a complex workflow executed on the Cray DataWarp, including simulation, analysis and visualization.

Neuromap – Replib on Cray Burst Buffer
Default parameters

- We have three cases, writing data in chunks of 100KB, 650KB, and 1MB
- We save the data in a shared file, each MPI process saves its own data and the size of the output file varies from 9 GB up to 400 GB
The maximum I/O efficiency is less than 19% for all the cases. We will tune the parameters for better performance.
### MPI I/O Statistics – Default parameters

For 32 nodes with default settings:

<table>
<thead>
<tr>
<th>Phase</th>
<th>min</th>
<th>max</th>
<th>ave</th>
</tr>
</thead>
<tbody>
<tr>
<td>file write time</td>
<td>22.92</td>
<td>23.58</td>
<td>23.25</td>
</tr>
<tr>
<td>time scale: 1 = 2**7 clock ticks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>523689105</td>
<td></td>
<td></td>
</tr>
<tr>
<td>imbalance</td>
<td>148522</td>
<td>248791</td>
<td>198657</td>
</tr>
<tr>
<td>local compute</td>
<td>4516667</td>
<td>4527122</td>
<td>4521894</td>
</tr>
<tr>
<td>wait for coll</td>
<td>1225864</td>
<td>7717977</td>
<td>4471921</td>
</tr>
<tr>
<td>collective</td>
<td>1092307</td>
<td>1149546</td>
<td>1120927</td>
</tr>
<tr>
<td>exchange/write</td>
<td>890825</td>
<td>908929</td>
<td>899877</td>
</tr>
<tr>
<td>data send</td>
<td>90654295</td>
<td>96061956</td>
<td>93358125</td>
</tr>
<tr>
<td>file write</td>
<td>412044716</td>
<td>423894304</td>
<td>417969510</td>
</tr>
<tr>
<td>other</td>
<td>568026</td>
<td>633007</td>
<td>600516</td>
</tr>
</tbody>
</table>

The data send bandwidth is quite slow because all the MPI processes send data to just two MPI I/O aggregators (2 BB nodes) and it takes 17% of the total time. The net write BW is 2231 MB/s because we have one MPI process per BB node that writes data.
In order to stress the SSDs, we increase the MPI I/O aggregators, according to our tests we can even disable the collective I/O. Optimization declaration for the case of 650KB:

```
MPICH_MPIIO_HINTS="$DW_JOB_STRIPED/out2*:romio_ds_write=disable:romio_cb_write=disable:striping_unit=665600"
```

The performance was improved up to 3.16 times.
The bottleneck of the data send does not exist anymore because each MPI process saves its data independent from the other ones aggregators (collective I/O is disabled) and the net write BW is almost 3x times faster than the default parameters.
Neuromap – Replib on Cray Burst Buffer
I/O efficiency comparison with optimized parameters

Similar, the I/O efficiency is improved maximum by 3.16 times.
Neuromap – Replib on Cray Burst Buffer

- We save the data in a shared file, each MPI process saves its own data and the size of the output file varies from 9 GB up to 12.5 TB
- We use 1 up to 268 BB nodes
- We achieve up to 0.5 TB/s with 4096 compute nodes and 268 BB nodes.
For the case of chunks of 100KB, the I/O efficiency is between 6.54% and 24.6%.

However, for the cases of 650KB and 1MB, the I/O efficiency varies from 29.7% till 56%.
Study-case WRF-CHEM (on Cori)
WRF-CHEM on Burst Buffer

- Weather Research and Forecasting Model coupling with Chemistry
- Small domain: 330 x 275
- Size of input file: 804 MB
- Size of output file: 2.9GB, it is saved every one hour of simulation
- Output file quite small
- For all the WRF-CHEM experiments we use 1280 MPI processes (40 nodes), as this is the optimum for the computation/communication
- For the default case, we stage-in all the files and we execute the simulation from BB
Total execution time and I/O on BB without MPICH_MPIIO_HINTS (default)

- The best total execution time is provided when we have 4 DW nodes.
- On Lustre the best execution time is with 64 OSTs and it is 15% faster than BB.
Darshan – WRFChem
1 BB node – default parameters
MPI I/O phases Statistics (MPICH_MPIIO_TIMERS=1) |

+--------------------------------------------------------------------------+
| MPIIO read by phases, readers only, for wrfinput_d01                      |
| min       max       ave                                        |
| file read = 1.54 1.54 1.54                                        |
| time scale: 1 = 2**6 clock ticks min max ave                      |
| total = 773580678                                                  |
| imbalance = 284814 284814 284814 0%                                |
| local compute = 91505804 91505804 91505804 11%                    |
| wait for coll = 2398813 2398813 2398813 0%                        |
| collective = 3646301 3646301 3646301 0%                           |
| read/exchange = 18196022 18196022 18196022 2%                     |
| file read = 55222120 55222120 55222120 7%                         |
| data receive = 588775983 588775983 588775983 76%                 |
| other = 12888553 12888553 12888553 1%                             |
| data receive BW (MiB/s) = 0.146                                     |
| raw read BW (MiB/s) = 1819.310                                    |
| net read BW (MiB/s) = 129.872                                    |
+--------------------------------------------------------------------------+

Timing for processing wrfinput file (stream 0) for domain 1: 21.68633 elapsed seconds
### MPI I/O phases Statistics

**MPICH_MPIIO_TIMERS=1**

<table>
<thead>
<tr>
<th>File write time scale: 1 = 2**7</th>
<th>min</th>
<th>max</th>
<th>ave</th>
</tr>
</thead>
<tbody>
<tr>
<td>file write time</td>
<td>2.30</td>
<td>2.30</td>
<td>2.30</td>
</tr>
</tbody>
</table>

**Time scale: 1 = 2**7:

<table>
<thead>
<tr>
<th>total</th>
<th>532124046</th>
</tr>
</thead>
<tbody>
<tr>
<td>imbalance</td>
<td>158972</td>
</tr>
<tr>
<td>local compute</td>
<td>48146033</td>
</tr>
<tr>
<td>wait for coll</td>
<td>855958</td>
</tr>
<tr>
<td>collective</td>
<td>1589992</td>
</tr>
<tr>
<td>exchange/write</td>
<td>9748711</td>
</tr>
<tr>
<td>data send</td>
<td>418919605</td>
</tr>
<tr>
<td>file write</td>
<td>41345308</td>
</tr>
<tr>
<td>other</td>
<td>10140527</td>
</tr>
</tbody>
</table>

**Data send BW (MiB/s)**: 0.107
**Raw write BW (MiB/s)**: 1262.748
**Net write BW (MiB/s)**: 98.114

---

**Timing for Writing wrfout_d01_2007-04-03_01_00_00 for domain 1**: 30.25151 elapsed seconds
MPI I/O phases Statistics (MPICH_MPIIO_TIMERS=1) III

+--------------------------------------------------------------------------+
| MPIIO read by phases, readers only, for wrfbdy_d01                      |
| min          max          ave                                      |
| file read time = 1.31 1.31 1.31                                      |
| time scale: 1 = 2**8  clock ticks  min  max  ave  ----------  ----------  ----------  --- |
| total = 995854066                                                   |
| imbalance = 349921  349921  349921  0%                               |
| local compute = 133146526  133146526  133146526  13%                |
| wait for coll = 1342000  1342000  1342000  0%                        |
| collective = 4455424  4455424  4455424  0%                          |
| read/exchange = 22374792  22374792  22374792  2%                    |
| file read = 11742536  11742536  11742536  1%                        |
| data receive = 803299250  803299250  803299250  80%                  |
| other = 18892054  18892054  18892054  1%                            |
| data receive BW (MiB/s) = 0.210                                     |
| raw read BW (MiB/s) = 271.116                                       |
| net read BW (MiB/s) = 3.197                                        |
+--------------------------------------------------------------------------+

Timing for processing lateral boundary for domain 1: 111.10603 elapsed seconds
Compare the total execution time on single DW nodes across various MPI I/O aggregators

- Example for declaring 4 MPI I/O aggregators
  
  ```
  export MPICH_MPIIO_HINTS="wrfinput*:cb_nodes=4,wrfout*:cb_nodes=4, wrbf*:cb_nodes=4"
  ```

- Tip: You can declare different MPI I/O aggregators per file
Understand the MPI I/O statistics on BB
(MPICH_MPIIO_STATS=1) I

<table>
<thead>
<tr>
<th>MPIIO read access patterns for wrfinput_d01</th>
</tr>
</thead>
<tbody>
<tr>
<td>independent reads = 1</td>
</tr>
<tr>
<td>collective reads = 527360</td>
</tr>
<tr>
<td>independent readers = 1</td>
</tr>
<tr>
<td>aggregators = 4</td>
</tr>
<tr>
<td>stripe count = 1</td>
</tr>
<tr>
<td>stripe size = 8388608</td>
</tr>
<tr>
<td>system reads = 762</td>
</tr>
<tr>
<td>stripe sized reads = 108</td>
</tr>
<tr>
<td>total bytes for reads = 2930104643 = 2794 MiB = 2 GiB</td>
</tr>
<tr>
<td>ave system read size = 3845281</td>
</tr>
<tr>
<td>number of read gaps = 2</td>
</tr>
<tr>
<td>ave read gap size = 0</td>
</tr>
</tbody>
</table>

We have 4 MPI I/O aggregators
We use one BB node (stripe count)
Default stripe size 8 MB
Only 14.17% of the reads are striped (100*108/762)
The average system read size is less than 4MB, the stripe size should be close to the average system read size

See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations.
Understand the MPI I/O statistics on BB
(MPICH_MPIIO_STATS=1) II

<table>
<thead>
<tr>
<th>MPIIO write access patterns for wrfout_d01_2007-04-03_00_00_00</th>
</tr>
</thead>
<tbody>
<tr>
<td>independent writes   = 2</td>
</tr>
<tr>
<td>collective writes    = 552960</td>
</tr>
<tr>
<td>independent writers  = 1</td>
</tr>
<tr>
<td>aggregators          = 4</td>
</tr>
<tr>
<td>stripe count         = 1</td>
</tr>
<tr>
<td>stripe size          = 8388608</td>
</tr>
<tr>
<td>system writes        = 797</td>
</tr>
<tr>
<td>stripe sized writes  = 114</td>
</tr>
<tr>
<td>aggregators active   = 234240,0,0,318720 (1, &lt;= 1, &gt; 1, 2)</td>
</tr>
<tr>
<td>total bytes for writes = 3045341799 = 2904 MiB = 2 GiB</td>
</tr>
<tr>
<td>ave system write size = 3821006</td>
</tr>
<tr>
<td>read-modify-write count = 0</td>
</tr>
<tr>
<td>read-modify-write bytes = 0</td>
</tr>
<tr>
<td>number of write gaps = 2</td>
</tr>
<tr>
<td>ave write gap size   = 4194300</td>
</tr>
</tbody>
</table>

Similar 14.3% of the writes are striped (100*114/797)

See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations.
Understand the MPI I/O statistics on BB (MPICH_MPIIO_STATS=1) III

+--------------------------------------------------------+
| MPIIO read access patterns for wrfbdy_d01               |
| independent reads = 2                                   |
| collective reads = 2338560                              |
| independent readers = 1                                 |
| aggregators = 2                                         |
| stripe count = 1                                        |
| stripe size = 8388608                                   |
| system reads = 1876                                     |
| stripe sized reads = 0                                  |
| total bytes for reads = 371398962 = 354 MiB             |
| ave system read size = 197973                           |
| number of read gaps = 6                                 |
| ave read gap size = 0                                   |
| See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations. |
+--------------------------------------------------------+

All the reads are not striped which mean this I/O is not efficient.
The average system read size is 197973 bytes.
Declaring MPICH MPIIO HINTS parameters based on the previous data

\[
\text{MPICH\_MPIIO\_HINTS=\textquoteleft\textquoteleft wrfinput*\textquoteleft\textquoteleft:cb\_nodes=4:striping\_unit=2097152, wrfout*\textquoteleft\textquoteleft:cb\_nodes=4:striping\_unit=2097152, wrfb*:cb\_nodes=4:striping\_unit=197973\textquoteright\textquoteright}
\]

| jobid: 6575384 | uid: 74747 | nprocs: 1280 | runtime: 172 seconds |

I/O performance estimate (at the MPI-Io layer): transferred 182112 MiB at 350.40 MiB/s

I/O performance estimate (at the STIO layer): transferred 0.2 MiB at 1.34 MiB/s

The execution time was decreased by almost 30%
Understand the MPI I/O statistics on BB
(MPICH_MPIIO_STATS=1) IV

+--------------------------------------------------------+
| MPIIO read access patterns for wrfinput_d01             |
| independent reads     = 1                           |
| collective reads      = 527360                       |
| independent readers   = 1                           |
| aggregators           = 4                           |
| stripe count          = 1                           |
| stripe size           = 2097152                      |
| system reads          = 1810                        |
| stripe sized reads    = 1141                        |
| total bytes for reads = 2930104643 = 2794 MiB = 2 GiB|
| ave system read size  = 1618842                      |
| number of read gaps   = 2                           |
| ave read gap size     = 0                           |
| See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations. |
+--------------------------------------------------------+

Timing for processing wrfinput file (stream 0) for domain 1: 9.56521 elapsed seconds

We have 4 MPI I/O aggregators
We use one BB node (stripe count)
New stripe size 2 MB
Only 63% of the reads are striped
The number of the operations increase (1810 reads)
Understand the MPI I/O statistics on BB (MPICH_MPIIO_STATS=1) V

+--------------------------------------------------------+
| MPIIO write access patterns for wrfout_d01_2007-04-03_00_00_00 |
| independent writes = 2                                      |
| collective writes  = 552960                                  |
| independent writers = 1                                     |
| aggregators       = 4                                       |
| stripe count      = 1                                       |
| stripe size       = 2097152                                  |
| system writes     = 1886                                    |
| stripe sized writes = 1183                                 |
| aggregators active = 208640,33280,0,311040 (1, <= 2, > 2, 4)|
| 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                                |
| See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations. |
+--------------------------------------------------------+

62.7% of the writes are striped

Timing for Writing wrfout_d01_2007-04-03_00_00_00 for domain 1: 12.99924 elapsed seconds
Understand the MPI I/O statistics on BB (MPICH_MPIIIO_STATS=1) VI

+-------------------------------------------------------------------------+
| MPIIIO read access patterns for wrfbdy_d01                              |
| independent reads = 2                                                   |
| collective reads = 2338560                                              |
| independent readers = 1                                                |
| aggregators = 4                                                        |
| stripe count = 1                                                       |
| stripe size = 197973                                                    |
| system reads = 3705                                                    |
| stripe sized reads = 114                                                |
| total bytes for reads = 371398962 = 354 MiB                            |
| ave system read size = 100242                                           |
| number of read gaps = 5                                                |
| ave read gap size = 444575572                                          |
| See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations.|
+-------------------------------------------------------------------------+

3% of the reads are striped

Timing for processing lateral boundary for domain 1: 83.90572 elapsed seconds
Looking for the optimum parameters

- We executed more experiments and tested various parameters according to the MPI IO statistics data.

- If the performance does not increase while we decrease the value of the striping unit, increase the number of the MPI I/O aggregators.

- While we decrease the value of the striping unit, the number of reads/writes is increasing. Maybe there is a need to use more BB nodes to achieve better performance.
The execution time was decreased by 57% on just one BB node!

Optimum parameters
MPICH_MPIIO_HINTS="wrfinput*:cb_nodes=16:striping_unit=262144,\wrfout*:cb_nodes=16:striping_unit=262144,\wrfb*:cb_nodes=16:striping_unit=50482"
Studying MPI I/O aggregators and striping size

<table>
<thead>
<tr>
<th>Parameters</th>
<th>I/O duration for wrfinput (in sec.)</th>
<th>I/O duration for wrfout (in sec.)</th>
<th>I/O duration for wrbdy_d01 (in sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>21.68</td>
<td>30.25</td>
<td>111.10</td>
</tr>
<tr>
<td>Optimized</td>
<td>5.51</td>
<td>7.27</td>
<td>32.8</td>
</tr>
</tbody>
</table>

The I/O bandwidth was improved between 3.4 and 4.1 times
Comparison between BB and Lustre on Shaheen

The total execution time on BB is 13.4% faster than Lustre for one hour of simulation of WRF-CHEM. For 24 hours of simulation the execution time on BB is faster than Lustre by 14.8%

We achieved better performance with BB by using one single BB node in comparison to 64 OSTs of Lustre
WRF-CHEM – Split output to one file per process

Reported “I/O” time from WRF-CHEM

Time (in seconds)

# DW nodes

0 0.5 1 1.5 2 2.5 3

1 2 4 8 16 32 64

File 2.9GB
WRF-CHEM – Split output to one file per process II

I/O efficiency using reported “I/O” time from WRF-CHEM

- File 2.9GB
WRF – Split output to one file per process – Large cases

Reported “I/O” time from WRF

WRF – Alaska domain 1km 6075 x 6075 x 28, 256 compute nodes
I/O efficiency using reported “I/O” time from WRF

- File 81GB
- File 361GB

Percentage (in %)

# DW nodes:
- 4
- 8
- 16
- 32
- 64
- 128

WRF – Split output to one file per process II
Multiple runs

- Submitting 3 jobs of 20 compute nodes and requesting 64 DW nodes each one
  - `used_bb_nodes.sh`
    - 192 BB nodes are used with at least one BB job
    - 0 BB nodes are used from more than one BB job
  - Variation 2-3%

- Variation can be significant when the system is more than 60-70% used
DataWarp vs Lustre for same number of nodes (OSTs)

WRF reports I/O time but it includes other functionalities which is beyond I/O
DataWarp vs Lustre, percentage of performance difference
WRF – Lustre vs DW

The graph compares the total execution time of WRF in Lustre and DW across different numbers of DW nodes. As the number of DW nodes increases, the total execution time decreases for both Lustre and DW, with Lustre consistently having a higher execution time than DW across all node counts.
Study-case Seissol
SeisSol I

- SeisSol is a software package for simulating wave propagation and dynamic rupture based on the arbitrary high-order accurate derivative discontinuous Galerkin method.

- Using 128 DataWarp nodes with 256 compute nodes. Developer provided an I/O kernel benchmark called checkpoint and it is available in the corresponding github repository.

- Many back-ends to be tests, MPI I/O, POSIX, HDF5, the SIONLIB had some issues.
SeisSol II

The developers have already integrated many advanced parameters such as:

SEISSOL_CHECKPOINT_ALIGNMENT=8388608
SEISSOL_CHECKPOINT_BLOCK_SIZE=8388608
SEISSOL_CHECKPOINT_SION_BACKEND=ansi
SEISSOL_CHECKPOINT_SION_NUM_FILES=1
SEISSOL_CHECKPOINT_SION_COLL_SIZE=0
SEISSOL_CHECKPOINT_CB_NODES=256
SEISSOL_CHECKPOINT_ROMIO_CB_WRITE=disable
SEISSOL_CHECKPOINT_ROMIO_DS_WRITE=disable
SEISSOL_CHECKPOINT_MPIO_LARGE_BUFFER=0
### SeisSol Results

<table>
<thead>
<tr>
<th>Filesystem</th>
<th>Back-end</th>
<th>I/O write performance (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lustre</td>
<td>MPI I/O</td>
<td>100</td>
</tr>
<tr>
<td>DataWarp</td>
<td>MPI I/O</td>
<td>472</td>
</tr>
<tr>
<td>DataWarp</td>
<td>POSIX</td>
<td>503</td>
</tr>
<tr>
<td>DataWarp</td>
<td>HDF5</td>
<td>449</td>
</tr>
</tbody>
</table>

In this case, DataWarp is 4.72 times faster than Lustre and around to 60% I/O efficiency.
DataWarp API

- Libatawarp
- dw_get_stripe_configuration
- dw_query_directory_stage
- dw_query_file_stage
- dw_set_stage_concurrency
- dw_stage_file_out
- dw_wait_directory_stage
ExpBB: An auto-tuning framework to explore the Performance of Burst Buffer (Cray DataWarp)
Motivation

- Burst Buffer (BB) does not provide the expected performance... or we do not know how to use it?
- A user should be familiar with some technical details and most of them are science-focus researchers.
- We need a tool that a user can execute and extract the optimized parameters for his application and the used domain.
• Fill in the required information in the beginning of the ExPBB script

• export executable="btio"

• #Declare option for the executable (leave empty if no arguments)
  export arguments="inputbt1.data"

• #Declare the minimum requested Burst Buffer size in GB
  export min_bb_size=1
Framework preparation II

- #Declare stage-in folder, full path
  
  export stage_in="/project/k01/markomg/development/expbb"

- #Declare stage-out folder, full path
  
  export stage_out="/project/k01/markomg/back2"

- The executable is required to have been compiled with the Darshan profiling tool

- The framework works for parallel I/O on shared file
Important MPI environment variables

- `export MPICH_ENV_DISPLAY=1`
  - Displays all settings used by the MPI during execution

- `export MPICH_VERSION_DISPLAY=1`
  - Displays MPI version

- `export MPICH_MPIIO_HINTS_DISPLAY=1`
  - Displays all the available I/O hints and their values

- `export MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1`
  - Display the ranks that are performing aggregation when using MI-I/O collective buffering

- `export MPICH_MPIIO_STATS=1 or 2`
  - Statistics on the actual read/write operations after collective buffering

- `export MPICH_MPIIO_HINTS="..."`
  - Declare I/O hints
Execution of ExPBB

- If your submission script is called btio.sh, then execute:

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

- Then the following will happen:
  - A parser will extract the compute resources from the original script and it will add the corresponding #DW commands in a copy of the original script. From the requested GBs the number of minimum BB nodes will be calculated.
  - The previous important MPI environment variables are added to all the new generated submission scripts.
  - Two executions will take place, one on Lustre and one on BB. This happens for two reasons, first to extract the basic execution time for comparison reasons, and second to extract the default striping unit and buffer for each case.
Execution of ExPBB II

- Then the tool will create a new submissions script depending on the number of the BB nodes, for example on Shaheen II we have 268 BB nodes, if we need 4 BB nodes minimum, then there will be scripts for 4, 8, 16, 32, 64, 128, and 256 BB nodes.

- Each of the script includes extra code before and after the `srun` command, where loops change the values of the parameters, where their range depends on the default values extracted on the first BB execution.

- After the `srun` command a parser is called, where it reads the Darshan performance data and acts accordingly.

- The first script will be submitted with the minimum requested nodes and it will start investigating the results.

- All the results will be written in txt files that are easily accessible.
#!/bin/bash

#SBATCH --partition=workq
#SBATCH --t 10
#SBATCH --A k1267
#SBATCH --ntasks=1024
#SBATCH --ntasks-per-node=32
#SBATCH --ntasks-per-socket=16
#SBATCH --J btio
#SBATCH --o btio_out_%j
#SBATCH --e btio_err_%j

srun -n 1024 --hint=nomultithread ./btio inputbt1.data
ExPBB example – Converted script

Script converted with ExpBB
Code not final, to be modified in the released version
Rules

- If the performance becomes worse while we decrease the striping unit and the number of system write/reads is significant large, then increase the MPI I/O aggregators. If the I/O is slower again, then restart with the used number of MPI I/O aggregators but initial parameters values.

- When the exploration of specific number BB nodes finish, submit another job with double BB nodes and compare with the previous best performance result.
Results I

The total execution time is improved 1.7 times with ExPBB and the I/O is improved up to 3.8 times for 1 BB node. Finally, the total execution time is 13.4% faster than Lustre with 64 OSTs.
The I/O was improved with ExPBB between 1.28 till 3.8 times. The execution on 16 BB nodes with ExPBB is faster than 64 BB nodes without ExpBB. MPICH_MPIIO_HINTS="wrf*:cb_nodes=128:striping_unit=4194304, wrf*:cb_nodes=256:striping_unit=4194304, wrfr*:cb_nodes=256:striping_unit=4194304"
We observe that for 8 BB nodes, with ExPBB framework, we have better performance than every other configuration. The maximum speedup compared to default BB execution, is 4.84. Moreover, 8 BB nodes have better performance than 64 OSTs. The ROI in this case is significant higher with Cray DataWarp.
I/O - Efficiency

Cray - DataWarp - I/O Efficiency

I/O efficiency (in %)

DW nodes

DataWarp - Default  DataWarp - ExpBB
./expbb btio.sh

Preparing and executing default script on Lustre

I/O duration for the file btio.mpi on Lustre with 1 OSTs is 155.26 seconds
I/O duration for the file btio.mpi on Lustre with 2 OSTs is 69.87 seconds
I/O duration for the file btio.mpi on Lustre with 4 OSTs is 35.27 seconds
I/O duration for the file btio.mpi on Lustre with 8 OSTs is 18.57 seconds
I/O duration for the file btio.mpi on Lustre with 16 OSTs is 10.12 seconds
I/O duration for the file btio.mpi on Lustre with 32 OSTs is 5.93 seconds
I/O duration for the file btio.mpi on Lustre with 64 OSTs is 5.59 seconds
I/O duration for the file btio.mpi on Lustre with 128 OSTs is 6.10 seconds
Preparing and executing default script on Burst Buffer

The I/O duration for the file btio.mpi on Burst Buffer with default parameters is 96.62 seconds

Starting auto-tuning execution on 1 Burst Buffer nodes

I/O duration for the file btio.mpi on 1 Burst Buffer with optimized parameters is 23.617 seconds

The new submission file with optimized parameters is named expbb_1_btio.sh

MPICH_MPIIO_HINTS=$DW_JOB_STRIPED/btio.mpi:cb_nodes=32:striping_unit=1048576:cb_buffer_size=4194304

Starting auto-tuning execution on 8 Burst Buffer nodes

I/O duration for the file btio.mpi on 8 Burst Buffer with optimized parameters is 4.99633 seconds

The new submission file with optimized parameters is named expbb_8_btio.sh

MPICH_MPIIO_HINTS=$DW_JOB_STRIPED/btio.mpi:cb_nodes=128:striping_unit=2097152:cb_buffer_size=8388608
New submission script for 1 BB node

```bash
#!/bin/bash

#SBATCH --partition=workq
#SBATCH --t 10
#SBATCH -A k1267
#SBATCH --ntasks=1024
#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=368GiB
#DW stage_in type=directory source=/project/k01/markomg development/expbb python_new/expbb python/ destination=$DW_JOB_STRIPED
#DW stage_out type=directory destination=/project/k01/markomg/back2 source=$DW_JOB_STRIPED

MPICH_MPIIO_HINTS=$DW_JOB_STRIPED/btio.mpi:cb_nodes=32:striping_unit=1048576:cb_buffer_size=4194304

cp tmp_inputbt1.data inputbt1.data
echo $DW_JOB_STRIPED >> inputbt1.data

srun --n 1024 --hint=nomultithread ./btio inputbt1.data
```
Study-case PIDX
PIDX

- PIDX is an efficient parallel I/O library that reads and writes multiresolution IDX data files
- It can provide high scalability up to 768k cores
- Successful integration with several simulation codes
  - KARFS (KAUST Adaptive Reacting Flow Solvers) on Shaheen II
  - Uintah with production runs on Mira
  - S3D

https://www.sci.utah.edu/software/pidx.html
PIDX description

File count are tunable (based on the file system in use).

- Synchronous reducing mode of communication using one-sided MPI
- To balance the data movement, the aggregators are uniformly placed across the ranks.
- File locking contention is reduced by having an aggregator to write to only one file.
The library achieves up to 900 GB/s on Burst Buffer, while we save 64 MB (2x32) per MPI process.
Efficiency based on IOR peak

Percentage of efficiency

# BB nodes
Complex Workflows
Case 1: WRF-CHEM
Outline

- Motivation
- In-depth explanation
- Demo - video
Motivation

- Using compute resources, while producing wrong results, costs time and money (even in electricity)
- Spending core-hours from team project
- You are not sure if the simulation has any issue
Study case – WRF-CHEM

- This is a real case of a ShaheenII user at KAUST.
- 40 compute nodes are used
- Around to 3GB of data are saved for specific time-steps.
Methodology

- First, we declare the required Burst Buffer (BB) space in persistent mode (*create_persistent.sh*).

- Then we start the execution of the model, using the BB persistent space.

- Then we start the execution of the tool *plot_and_stage_out.sh* that does the following:
  - Check the existence of any output file (we know the filename pattern).
  - When an output file exists (NetCDF format), we use a script in Python with NetCDF and Matplotlib libraries to read the output file and save one variable to an image file (with same filename pattern).
  - Then a tool which uses DataWarp API, stages out *only* the image into the Lustre parallel filesystem.

- The same moment with the *plot_stage_out.sh*, we execute the *wait.sh* script which runs on the login node. This script recognizes when an image has been stage-out and it visualizes it for the user. Then, the user observes if the simulation is correct or not and can stop the simulation if it is required.

Instructions here:  
https://github.com/gmarkomanolis/bb_ixpug18  
folder: complex_workflow/persistent_vis
Creating Persistent BB allocation

- File: create_persistent.sh
- Execution: sbatch create_persistent.sh

```bash
#!/bin/bash -x
#SBATCH --partition=workq
#SBATCH -t 1
#SBATCH -A k01
#SBATCH --nodes=1
#SBATCH -J create_persistent_space

#BB create_persistent name=george_test capacity=600G access=striped
type=scratch
exit 0
```
Executing the main application I

- File: `wrfchem_bb_persistent.sh`
- Execution: `sbatch wrfchem_bb_persistent.sh` (check the job id)

```bash
#SBATCH --partition=workq
#SBATCH -t 60
#SBATCH -A k01
#SBATCH --ntasks=1280
#SBATCH --ntasks-per-node=32
#SBATCH -J WRF_CHEM_PERSISTENT
#SBATCH -o out_%j
#SBATCH -e err_%j

#DW persistentdw name=george_test
#DW stage_in type=directory
source=/project/k01/.../forburst destination=${DW_PERSISTENT_STRIPED_george_test}

export MPICH_ENV_DISPLAY=1
export MPICH_VERSION_DISPLAY=1
export MPICH_MPIIO_HINTS_DISPLAY=1
export MPICH_STATS_DISPLAY=1`
```
export MPICH_MPIIO_HINTS="$DW_PERSISTENT_STRIPED_george_test/wrfinput*:*cb_nodes=40:striping_unit=131072,
$DW_PERSISTENT_STRIPED_george_test/wrfout*:*cb_nodes=40:striping_unit=65536"
export MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1
export MPICH_MPIIO_STATS=2

cd $DW_PERSISTENT_STRIPED_george_test
chmod +x wrf.exe

time srun -n 1280 --hint=nomultithread wrf.exe
Create an image of the output NetCDF file

- File: `plot_persistent.sh`
- Execute: `./plot_persistent.sh filename_netcdf`

```python
#!/.../python
import matplotlib
matplotlib.use('Agg')

import matplotlib.pyplot as plt
import netCDF4
import sys
nc = netCDF4.Dataset(str(sys.argv[1]))

# read all the data
topo = nc.variables['T2'][::1,::1]

# make image
plt.figure(figsize=(10,10))
plt.imshow(topo.squeeze(), origin='lower')

#plt.title(nc.title)
output=str(sys.argv[1])+'.png'
plt.savefig(output, bbox_inches=0)
```
Stage out using DataWarp API

- **File:** `stage_out.c`
- **Compile:**
  - `module load datawarp`
  - `cc -o stage_out stage_out.c`

```c
#include <stdio.h>
#include <datawarp.h>

int main(int argc, char **argv)
{
    char *infile, *outfile;
    int stage_out;
    infile = argv[1];
    outfile = argv[2];
    stage_out = dw_stage_file_out(infile, outfile, DW_STAGE_IMMEDIATE);
    return 0;
}
```
- **Execute:**
  - `srun -n 1 stage_out $DW_PERSISTENT_STRIPED_george_test/filename.png /project/k01/markomg/wrfchem_stage_out/filename.png`
Script to plot and stage out I

- File: `plot_stage_out.sh`
- Execute: `sbatch --dependency=after:app_job_id plot_stage_out.sh`

```bash
#!/bin/bash

#SBATCH --partition=workq
#SBATCH -t 30
#SBATCH -A k01
#SBATCH --ntasks=32
#SBATCH --ntasks-per-node=32
#SBATCH --ntasks-per-node=32
#SBATCH -J PLOT_AND_STAGE_OUT
#SBATCH -o out_%j
#SBATCH -e err_%j

#DW persistentdw name=george_test
#DW stage_in type=directory
source=/project/k01/markomg/burstbuffer/complexstage_in_bb/ destination=$DW_PERSISTENT_STRIPED_george_test
```
module load python/2.7.11
cd $DW_PERSISTENT_STRIPED_george_test
chmod +x plot_persistent.sh
chmod +x stage_out

let i=0
while [ $i -lt 24 ]
do
k=$(printf %02d $i)
if [-f wrfout_d01_2007-04-03_${k}_00_00 ]; then
  check_lsof=`lsof wrfout_d01_2007-04-03_${k}_00_00 | wc -l`
  while [ $check_lsof -eq 2 ]
do
    sleep 30
    check_lsof=`lsof wrfout_d01_2007-04-03_${k}_00_00 | wc -l`
done
  ./plot_persistent.sh wrfout_d01_2007-04-03_${k}_00_00
  srun -n 1 stage_out $DW_PERSISTENT_STRIPED_george_test/wrfout_d01_2007-04-03_${k}_00_00.png
  /project/k01/markomg/wrfchem_stage_out/wrfout_d01_2007-04-03_${k}_00_00.png
let i=$i+1
else
  sleep 30
fi
done
Visualize images when they arrive on the Lustre

- File: **wait.sh**
- Execute: `.wait.sh number_of_images /path_to_Lustre_stage_out_folder/`

```bash
#!/bin/bash

let i=0
while [ $i -lt $1 ]
do
  if [ -f $2/wrfout_d01_2007-04-03_$(printf "%02d" $i)_00_00.png ]; then
display $2/wrfout_d01_2007-04-03_$(printf "%02d" $i)_00_00.png &
  let i=i+1
  sleep 15
  else
    sleep 60
  fi
done
```
Delete Persistent BB allocation

- File: `delete_persistent.sh`
- Execution: `sbatch delete_persistent.sh`

```bash
#!/bin/bash
#SBATCH --partition=workq
#SBATCH -t 1
#SBATCH -A k01
#SBATCH --nodes=1
#SBATCH -J delete_persistent_space

#BB destroy_persistent name=george_test
exit 0
```
Case 2: In situ processing and visualization (collaboration with KVL)
Extremely Severe Cyclonic Storm Chapala was the second strongest tropical cyclone on record in the Arabian Sea, according to the American-based Joint Typhoon Warning Center (JTWC). The third named storm of the 2015 North Indian Ocean cyclone season, it developed on 28 October off western India from the monsoon trough. Fueled by record warm water temperatures, the system quickly intensified and was named Chapala by the India Meteorological Department (IMD). By 30 October, the storm developed an eye in the center of a well-defined circular area of deep convection. That day, the IMD estimated peak three-minute sustained winds of 215 km/h (130 mph), and the JTWC estimated one-minute winds of 240 km/h (150 mph); only Cyclone Gonu in 2007 was stronger in the Arabian Sea.
We execute Inshimtu and WRF on the same nodes (Inshimtu uses only the last core), one extra node for the post-process.

When a NetCDF file is written, then it is converted to VTK format but only the area that we are interested in, so we save less data.

In our largest case, by removing variables that we do not need and chopping specific area, from 28.2TB of NetCDF files, we save on Lustre 97GB.

Files are downloaded and visualized.
Results

- We use two domains, one small (1100x1000x34) and one larger (3500x3000x34).

- In order to increase the details in the available data, we are testing two cases, saving data every one hour and every 10 minutes.

- A post-processing tool chops from the whole area only the cyclone region and saves this file on BB.

Videos here:
https://github.com/gmarkomanolis/bb_ixpug18
folder: complex_workflow/cyclone_vis
Visualization

- Executing the simulation on Burst Buffer and save data every 10 minutes with manual tuning (6x times more data). Total execution time is 5% faster than Lustre.
Conclusions

- Using Burst Buffer is not difficult but achieving significant performance requires some effort.
- Burst Buffer boosts the performance for many demonstrated applications.
- Many parameters need to be investigated for the optimum performance.
- CLE 6.0 solves some BB issues but still needs optimizations.
- Implementing a complex workflow has several steps and it could combine persistent allocation, multiple applications having access to the same files, external scripts to handle the same files, and DataWarp API.
- Think clever and innovative on how to implement your workflow.
Thank you!
Questions?

gregorios.markomanolis@kaust.edu.sa
saber.feki@kaust.edu.sa