An I/O Study of ECP Applications

Chen Wang
Elena Pourmal
cwang@hdfgroup.org
epourmal@hdfgroup.org

1 INTRODUCTION

We studied and analysed the I/O patterns of four ECP applications and five HACC-IO benchmarks. Table 1 gives brief descriptions of those applications.

In this paper, we describe in details the steps of analyzing and tuning the HACC-IO benchmarks. We illustrate the impact of different access patterns, stripe settings and HDF5 metadata. We also compare the five benchmarks on two different parallel file systems, Lustre and GPFS. We show that HDF5 with proper optimizations can catch up the pure MPI-IO implementations.

Another goal of this paper is to understand the I/O behaviour of ECP applications and provide a systematic way to profiling and tuning the I/O performance. We mainly used two I/O profiling tools, Darshan [3] and Recorder [6] to conduct this study. We also made suggestions for each application on how to avoid undesired behaviours and how to further improve the I/O performance. In Section 4, we discuss the observations of for ECP applications.

1.1 Summary of observations and suggestions

We summary below the observations and some unexpected behaviours we found for each application along with the suggestions on how to fix them. Detailed results and analysis can be found in Section 3 and Section 4.

- **FLASH**: Unnecessary HDF5 metadata operations `H5Acreate()`, `H5Aopen()` and `H5Aclose()` are used during every checkpointing step. Those operations can be expensive especially when running a large number of iterations. This can be easily fixed at the price of losing some code modularity.
- **NWChem**: File-per-process patterns are found for writing local temporary files. This is undesired and will cause a lot of pressures on parallel file systems for large scale runs. Conflicting patterns are found for the runtime database file, which can lead to consistency issues when running on non-POSIX file systems.
- **Chombo**: The Same file-per-process pattern is observed for Chombo too. Moreover, Chombo by default uses independent I/O to write the final result to a shared HDF5 file. Depends on the problem scale and underlying file system configurations, collective I/O can be enabled to further optimize the I/O performance.
- **QMCPack**: One unexpected pattern is found for checkpoint files. QMCPACK overwrites the same checkpoint file for each computation section. This can lead to an unrecoverable state if a failure occurred during the checkpointing step.
- **HACC-IO**: HDF5 can use different data layout to achieve similar MPI-IO access patterns. Stripe settings of the parallel file system has a big impact on the write performance. Also the default metadata header can greatly slow down the write performance. However, carefully setting the alignment or metadata data block size, HDF5 can deliver a similar performance as the pure MPI-IO implementation.

In this paper, we use HACC-IO benchmarks as detailed example to illustrate the process of analysing and tuning I/O performance. In next section, we first introduce the five HACC-IO benchmarks we created for this study and then describe the access patterns exhibited by each of the benchmark. In Section 3, we present the tuning parameters we explored and the impact of them on I/O performance.

2 HACC-IO BENCHMARKS

In this section, we describe the five benchmarks we created for this study and the three access patterns exhibited by them. The same access patterns can be found in other scientific applications too. So some general advises and tuning methodologies should apply to other applications as well.

In all benchmarks, all processors write 9 variables to a single shared file and each variable has an identical size. Except for one benchmark (which will discuss later), all variables are stored together in an one dimensional array where each element in the array is a double-precision floating point value. The first two benchmarks are called *MPI Contiguous* and *MPI Interleaved* and they are implemented using pure MPI-IO. These two benchmarks serve as the baseline for comparison with the HDF5 implementations. As the names suggested, in MPI Contiguous benchmark, each processor writes each variable contiguously in the file whereas in MPI Interleaved benchmark, each variable is written interleaved. The code for writing variables is shown in Figure 1. The only difference between them is how to calculate the offset for the next write. In MPI Interleaved benchmark, the next write starts from where the current write finished. In MPI Contiguous benchmark, the next write starts from the current offset plus the variable size. Their access patterns are shown in Figure 4(a) and (b). From the view a local processor, the nine writes are contiguous. However, from the perspective of each variable, the writes are interleaved (in fact they are evenly strided). Also note that Figure 1 shows only the code for independent I/O for simplicity. Collective I/O is implemented using `MPI_File_write_at_all` instead of `MPI_File_write_at`.

The rest three benchmarks are implemented using the HDF5 library, namely *HDF5 Individual*, *HDF5 Multi* and *HDF5 Compound*.

HDF5 Individual benchmark uses the most common way to write multiple variables, with each variable as an individual dataset in the HDF5 representation. In the end, the output HDF5 file has one root group which contains nine separate dataset. The I/O part of the code is shown in Figure 2. This benchmark achieves the same access pattern as MPI Contiguous, as shown in Figure 4(a).

Table 1: List of studied applications

#	App	Version	Description
1	Flash [1]	4.4	A component-based multi-physics scientific simulation software package.
2	NWChem [5]	6.8.1	Open source high-performance computational chemistry.
3	Chombo [2]	3.2.7	Software for adaptive solutions of partial differential equations.
4	QMCPACK [4]	3.7.0	Electronic structure code that implements numerous Quantum Monte Carlo (QMC) algorithms.
5	HACC-IO	1.0	Five HACC I/O benchmarks adapted from GenericIO (https://xgitlab.cels.anl.gov/hacc/genericio).

```
// Benchmark 1. MPI Interleaved
for (i=0; i < NUM_VARS; i++) {
     MPI_File_write_at(fh, mpi_off, &writedata[i*NUM_DOUBLES_PER_VAR_PER_RANK], NUM_DOUBLES_PER_VAR_PER_RANK,
                       MPI_DOUBLE, &mpi_stat);
     mpi_off += BUF_SIZE_PER_VAR/mpi_size;     // Move to the end of the current write.
}

// Benchmark 2. MPI Contiguous
for (i=0; i < NUM_VARS; i++) {
     MPI_File_write_at(fh, mpi_off, &writedata[i*NUM_DOUBLES_PER_VAR_PER_RANK], NUM_DOUBLES_PER_VAR_PER_RANK,
                       MPI_DOUBLE, &mpi_stat);
     mpi_off += BUF_SIZE_PER_VAR;              // Move to the position of next variable
}
```

Figure 1: The code of writing nine variables in two MPI benchmarks.

HDF5 Multi benchmark has the same data representation as HDF5 Individual, and they also share the same code for I/O (Figure 2). The difference exists only in the dataset creation phase. HDF5 Multi uses an under-development API for creating the nine dataset. This new API tells the HDF5 library to use a different data layout to store the dataset, which can result in a MPI Interleaved access pattern as shown in Figure 4(b).

HDF5 Compound is different from the above-mentioned four implementations. In this benchmark, each process holds one data structure that stores all nine local variables. In HDF5, it is also called the compound datatype. The creation code is shown in Figure 3. With this representation, each processor performs only one write during the I/O phase, which is in contrast of nine writes required by the other four benchmarks.

3 HACC-IO EXPERIMENTS

3.1 Methodology

There are many tuning parameters one can explore to optimize I/O performance. It is unrealistic to try every combination. In this study, we fix the scale and size of our problem and focus more on tuning HDF5 to catch up the MPI baseline. To further narrow down the analysis, we only compare the time for the actual I/O functions of different benchmarks, i.e., we do not include the time for open, close, fsync and other operations. To be specific, for two MPI benchmarks, we measure the time spent on `MPI_File_write_at` (or `MPI_File_write_at_all` for collective I/O) function and for HDF5 benchmarks, we timing only the `H5Dwrite` function. We implemented HDF5 benchmarks in a way that metadata writes occurred only after the dataset has been written. So the comparison is fair as both implementations write the same amount of data. Finally, we report the write bandwidth for easy comparison. The bandwidth is calculated by: $BW = S/(t_e - t_s)$, where S is the total size of nine variables and t_e and t_s are the latest I/O finish time and the earliest I/O start time.

One guiding principle we tried to follow throughout the experiments is to keep the interference due to different tuning parameters at a minimum level. In other words, we tried to vary only one tuning knob at a time to see the impact of it. In the following sections, we will start by testing the collective I/O and then move on to parallel file system settings. Next, we report the performance slow down caused by HDF5 metadata and give suggestions on how to mitigate it. Finally, we show that with proper optimizations HDF5 can catch up to the pure MPI-IO performance.

3.2 Setup

Experiments are conducted on Quartz and Lassen at LLNL. Each compute node of Quartz is equipped with an Intel Xeon E5-2695 CPU which has 36 cores. All nodes are connected via Intel Omni-Path. Lassen is similar to the Sierra System. The CPU of Lassen's compute node is IBM Power 9 with 44 cores. Another major difference between the two systems and also the reason we choose these two systems is that they deployed two different parallel file systems: Lustre on Quartz and GPFS on Lassen.

Since two systems have different CPU architectures, they also come with their own vendor MPI implementations. Therefore, on Quartz all benchmarks are compiled with Intel MPI and on Lassen they are compiled with IBM MPI. The HDF5 we used is an internal and in-develop version (1.13), which provides the new Multi API.

As mentioned before, the problem size and scale are fixed. Each variable is 8GB, i.e., 1M doubles. The total file size (ignoring metadata and alignment) is thus 72GB. We run all experiments on 32 compute nodes with 32 MPI processes per node - a total of 1024 MPI processes. All variables are distributed evenly across all MPI

```
// Benchmark 3, 4: HDF5 Individual and HDF5 Multi
double *writedata = (double*) malloc(BUF_SIZE_PER_VAR/mpi_size*NUM_VARS);
for (i = 0; i < NUM_VARS; i++) {
     dset_ids[i] = H5Dopen(file_id, DATASETNAME[i], H5P_DEFAULT);
     mem_space_ids[i] = H5Screate_simple(1, mem_dims, NULL);
     file_space_ids[i] = H5Screate_simple(1, file_dims, NULL);

     // Select column of elements in the file dataset
     file_start[0] = mpi_rank * NUM_DOUBLES_PER_VAR_PER_RANK;
     file_count[0] = NUM_DOUBLES_PER_VAR_PER_RANK;
     H5Sselect_hyperslab(file_space_ids[i], H5S_SELECT_SET, file_start, NULL, file_count, NULL);

     // Select elements in the memory buffer
     mem_start[0] = i * NUM_DOUBLES_PER_VAR_PER_RANK;
     mem_count[0] = NUM_DOUBLES_PER_VAR_PER_RANK;
     H5Sselect_hyperslab(mem_space_ids[i], H5S_SELECT_SET, mem_start, NULL, mem_count, NULL);

     H5Dwrite(dset_ids[i], H5T_NATIVE_DOUBLE, mem_space_ids[i], file_space_ids[i], dxfer_plist_id, writedata);
}
```

Figure 2: The code of writing nine dataset in HDF5 Individual and HDF5 Multi benchmarks

```
typedef struct {
    double id;
    double mask;
    double x;
    double y;
    double z;
    double vx;
    double vy;
    double vz;;
    double phi;
} hacc_t;
```

```
// Benchmark 5: HDF5 Compound datatype
Hmemtype = H5Tcreate (H5T_COMPOUND, sizeof (hacc_t));
           H5Tinsert (Hmemtype, "id", HOFFSET (hacc_t, id), H5T_NATIVE_DOUBLE);
           H5Tinsert (Hmemtype, "mask", HOFFSET (hacc_t, mask), H5T_NATIVE_DOUBLE);
           H5Tinsert (Hmemtype, "x", HOFFSET (hacc_t, x), H5T_NATIVE_DOUBLE);
           H5Tinsert (Hmemtype, "y", HOFFSET (hacc_t, y), H5T_NATIVE_DOUBLE);
           H5Tinsert (Hmemtype, "z", HOFFSET (hacc_t, z), H5T_NATIVE_DOUBLE);
           H5Tinsert (Hmemtype, "vx", HOFFSET (hacc_t, vx), H5T_NATIVE_DOUBLE);
           H5Tinsert (Hmemtype, "vy", HOFFSET (hacc_t, vy), H5T_NATIVE_DOUBLE);
           H5Tinsert (Hmemtype, "vz", HOFFSET (hacc_t, vz), H5T_NATIVE_DOUBLE);
           H5Tinsert (Hmemtype, "phi", HOFFSET (hacc_t, phi), H5T_NATIVE_DOUBLE);

mem_dims[0] = NUM_HDATA_PER_RANK;
file_dims[0] = mem_dims[0]*mpi_size;
file_space_id = H5Screate_simple(1, file_dims, NULL);
dset_id = H5Dcreate2(file_id, "ALLVAR", Hmemtype, file_space_id, H5P_DEFAULT, dcpl_id,
H5P_DEFAULT);
```

Figure 3: The code of creating the compound datatype

ranks so each processor holds 8MB for each variable. In total each processor attributes 72MB data to the output file.

3.3 Results

3.3.1 Collective vs. Independent. Collective I/O is an important optimization in that it combines I/O requests into bigger blocks so that reads/writes to the I/O system will be larger. It is especially useful when the writes are small or non-contiguous. Combing non-contiguous I/O into contiguous I/O can effectively reduce the disk seek time. Also, collective I/O reduces the total number of I/O requests that can reduce the pressure on parallel file systems.

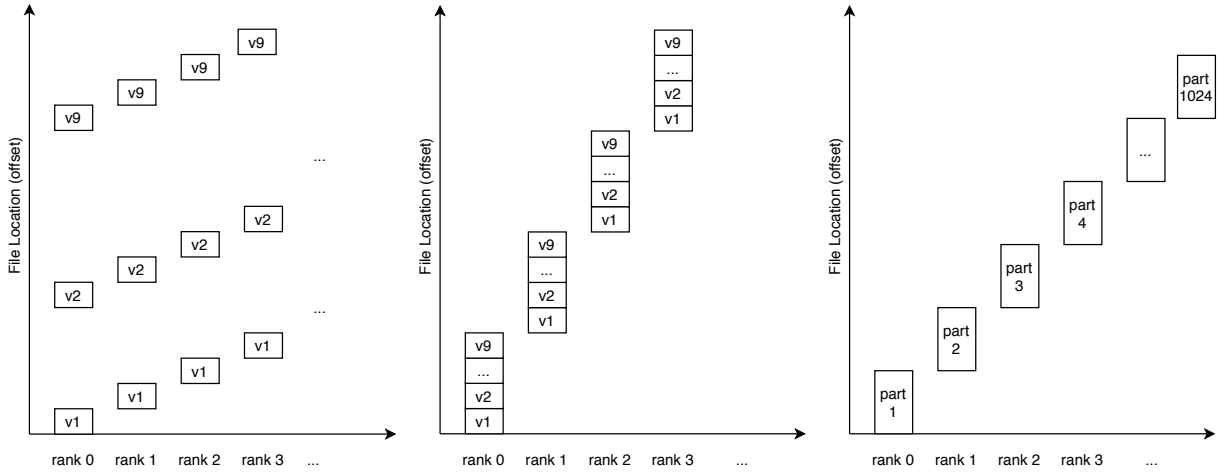
However, collective I/O is not always advantageous because sometimes the overhead of collective calls outweighs their benefits. And this is also what we observed in our case. In our benchmarks, each write is at least 8MB (72MB for HDF5 Compound) which is big enough to amortize the disk seek time. Since the file is striped into multiple storage targets, the writes to one target are already contiguous. As we can see from Table 2, collective I/O still significantly reduced the total I/O time, from 1346 seconds to 70 seconds. However, the cost for collective calls is too high - we spent over

Collective	Total time (s)
MPI_write_at_all	4354.71
write	70.31
Independent	**Total time (s)**
MPI_write_at	1346.97
write	1346.79

Table 2: Function cost of collective I/O and independent I/O

4,000 seconds on `MPI_File_write_at_all`. These results are collected from the MPI Contiguous benchmark running on Lustre. Other benchmarks have similar results. Thus, in the following experiments, we use only the independent I/O.

3.3.2 Impact of stripe size. This experiment is conducted only on Lustre since on GPFS users are not able to change the stripe settings. We fixed the stripe count to 32 which is the number of compute nodes we use. The performance against different stripe sizes is shown in Figure 5. We show here only the HDF5 benchmarks to



(a) MPI Contiguous / HDF5 Individual **(b) MPI Interleaved / HDF5 Multi** **(c) HDF5 Compound**

Figure 4: Assume no HDF5 metadata operations, the five HACC-IO benchmarks exhibit three different access patterns. MPI Contiguous and HDF5 Individual have the same access patterns (a). The second access pattern (b) can be achieved by MPI Interleaved or HDF5 Multi. In the last one, each process performs only one write and this access pattern is shown only by HDF5 Compound.

make the graph less crowded; the comparison between HDF5 benchmarks and MPI benchmarks is given in later sections.

On Lustre, the best performance is achieved by using HDF5 Individual and setting the stripe size to 128MB. One interesting finding is that HDF5 Individual and HDF5 Multi does not perform well on 36M stripe size. In that case, HDF5 Compound can deliver a higher write bandwidth. We use HDF5 Multi as the example to analyze the access pattern. Again, ignoring the metadata data, the access pattern of HDF5 Multi looks like Figure 6, which is exactly the same as MPI Interleaved. In this access pattern, each process writes 72MB, a half of it goes to one storage target i and the other half goes to another storage target $i + 1$. Since all writes are blocking calls, writes to v_j can not start until writes to v_{j-1} has finished. So if we look at the first 1024 writes to v_1, we can see that they only utilized a half of available storage targets, i.e., $1, 3, 5, ..., 31$. Moreover, the 36MB stripe size can not be divided by the write size (8MB), so the fifth variable on each process will be split into two different storage targets. Things become even worse when there is a metadata header at the beginning of the file, which we will discuss in next section.

Overall, the best strip setting depends on the access patterns and also the problem size and problem scale. Once the access pattern is known, we can perform a similar analysis to get the optimal stripe settings.

3.3.3 Impact of the metadata header. Figure 7 and Figure 8 depict the HDF5 access patterns that include a 2KB header. This 2KB header is reserved by default by HDF5 to keep metadata information. Due to this header, now in Figure 8, not only v_5, but also v_9 will be split into two different storage targets.

To study the exact impact of an offset gap on the performance, we modified the MPI benchmarks to start from a specific offset instead of zero. The outcomes are two folds: (1) it helps us understand how much performance penalty we pay for the 2KB header. (2) we

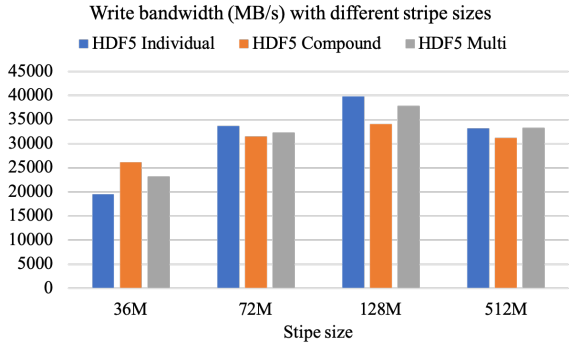


Figure 5: Write bandwidth on Lustre with different stripe sizes

can decide a better gap size to use as the alignment size or metadata block size for HDF5. The results are given in Figure 9 and Figure 10 for Lustre and GPFS respectively. We use the best benchmark (Section 3.3.4) for each file system to conduct the experiments, i.e., MPI Contiguous on Lustre and MPI Interleaved on GPFS. It can be seen that, offset gaps have a more severe impact on GPFS than Lustre. In both systems, a 2KB gap is certainly bad for the performance. Also on both systems, it seems that starting from 32MB or 128MB can deliver a even better performance than starting from zero. The reason for this needs further exploration and we leave it as our further work.

With this information, we are able to optimize HDF5 benchmarks to hide or mitigate the penalty of the metadata header. There are three ways to achieve such a goal:

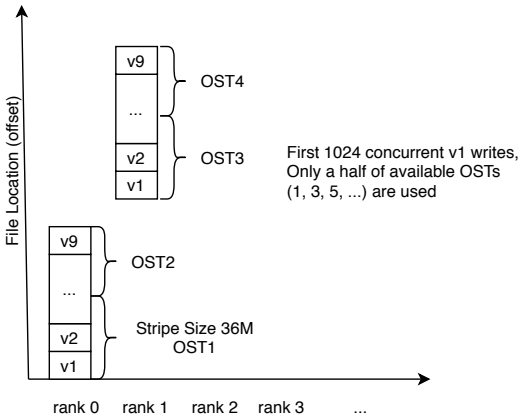


Figure 6: MPI Interleaved with 36MB stripe size

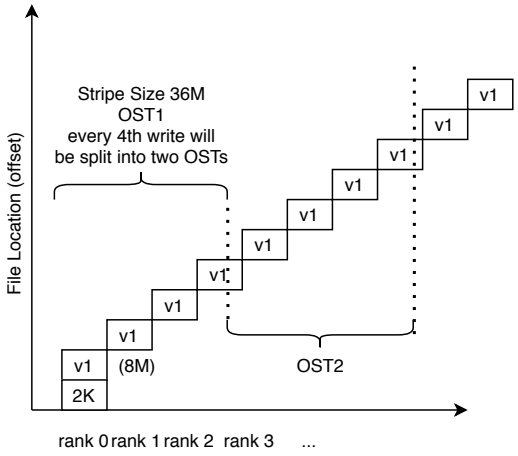


Figure 7: HDF5 Individual with 36MB stripe size

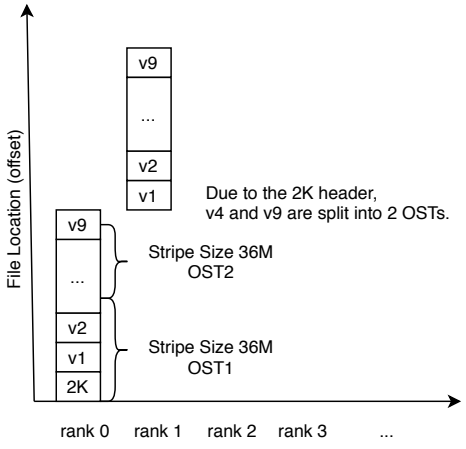


Figure 8: HDF5 Multi with 36M stripe size

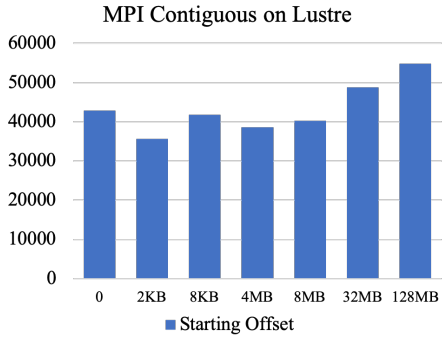


Figure 9: Impact of header on Lustre

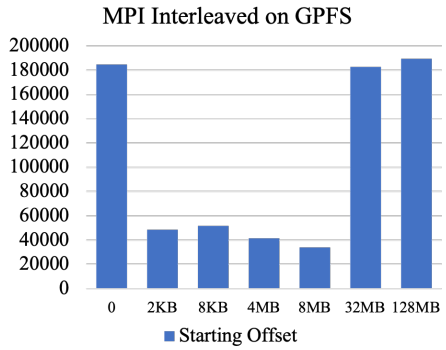


Figure 10: Impact of header on GPFS

- Use a split driver, which outputs metadata and raw data into two separate files. It requires minor codes change but it produces two files which may not be what users want.
- Use `H5Pset_alignment`. This call sets an alignment for every data write. For example, we could set a 32MB alignment to make sure the first data block starts from 32MB. However, there is another issue that this method can not handle: if the metadata exceeds 2KB, they probably will be scattered across the file, causing "holes" between data chunks.
- Use `H5Pset_meta_block_size`. This API changes the default reserved size for the metadata block. By setting a larger and better size, we can fit all metadata into the begging of the file and at the same time, the raw data will start from a desired offset. We use this method in next section.

3.3.4 Comparison of five benchmarks. So far we have tested with collective I/O, independent I/O, different stripe sizes and different header sizes. In this section, we mainly have two objectives: (1) find the best I/O access pattern for two different parallel file systems; (2) Optimize HDF5 to catch up the MPI implementations.

As we discussed from last section, HDF5 metadata header (2KB offset gap) could greatly hurt the performance. However, we can use `H5Pset_meta_block_size` to reduce this performance penalty. For different parallel file system systems, this metadata block size is not always the same. The metadata block size should be as small as possible since it introduces wasted space in the file. We set it

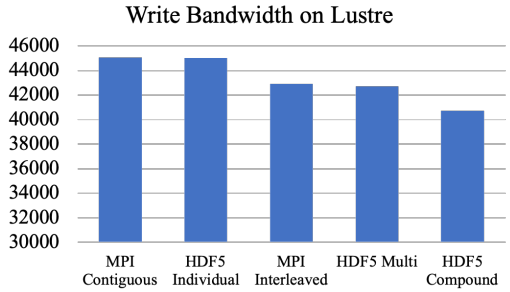


Figure 11: Write bandwidth on Lustre. The metadata block size is 8M for HDF5.

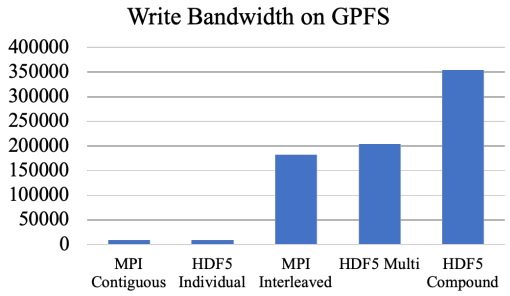


Figure 12: Write bandwidth on GPFS. The metadata block size is 32M for HDF5.

to 8MB on Lustre and 32MB on GPFS according to Figure 9 and Figure 10.

The results are shown in Figure 11 and Figure 12. For Lustre experiments we set the stripe size to 128MB as it gives the best performance as discussed in Section 3.3.2. Again, with this strip setting, MPI Contiguous delivers the best performance and with a 8MB metadata block size, HDF5 Individual is able to achieve a similar performance. On GPFS, we are not able to change the stripe size (or block size), which is 32MB on Lassen. With such a small stripe size, MPI Interleaved out performs MPI Contiguous. HDF5 Multi has exactly the same access pattern as MPI Interleaved. With a 32MB metadata block size, it even achieves better write bandwidth than MPI Interleaved. The improve over MPI Interleaved is mainly due to the 32MB offset shift as mentioned in Section 3.3.3.

4 ECP APPLICATIONS

4.1 Setup

All four applications are compiled with Intel Compiler 2019 and Intel MPI 2018. QMCPACK and HACC-IO uses HDF5 1.12.0 whereas FALSH and Chombo are compiled with HDF5 1.8.20. NWChem uses only POSIX for I/O. All experiments are conducted on Quartz at LLNL. Unless specified otherwise, all applications run on 8 nodes with 8 processes per node. The stripe count and stripe size of Lustre are 8 and 4MB respectively.

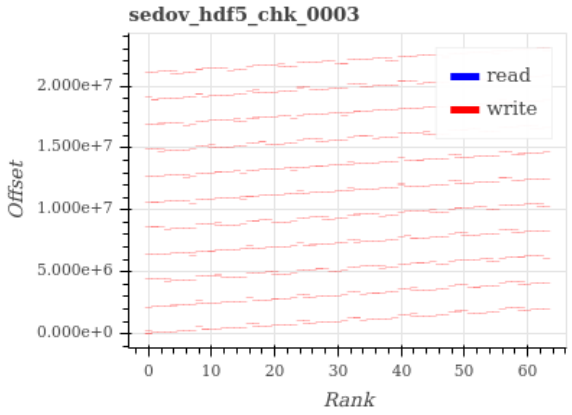


Figure 13: FLASH: access pattern of independent I/O

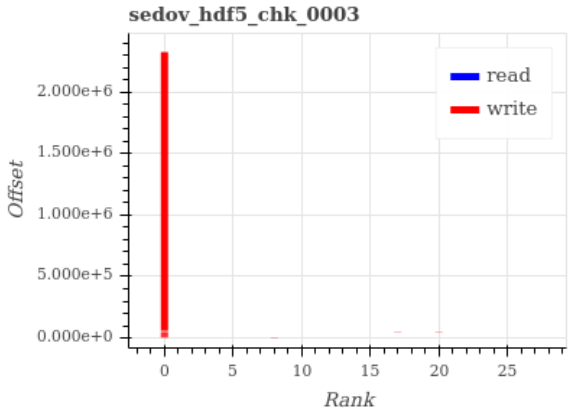


Figure 14: FLASH: access pattern of collective I/O

4.2 FLASH

The case we run is a 2D Sedov explosion simulation, with an unified grid size 512 × 512. We run it for 100 iterations and output checkpoints at a frequency of 20 iterations. Figure 14 shows the access patterns of one checkpoint file written using independent I/O. In this pattern, each rank writes its local data to a shared HDF5 file, which naturally leads to a large number of small writes. In contrast, Figure 13 shows the access pattern with collective I/O enabled, where rank 0 handles all data writes and the metadata are written in a round-robin matter (default behaviour of HDF5). Note that even though we set the stripe count to 8, ROMIO still decides to use only one rank as the aggregator.

Figure 15 shows the count of the most visited functions. We found that the number of `H5Aclose()` call is twice as that of `H5Aopen()`, and there is never a `H5Aread()` call. This is a strange pattern because normally we would create an attribute, then write to it and close it at the end. So the number of create, write and close should be the same. But in FLASH, in order to main code modularity, the attributes are created and closed first. Then at a latter time, they are opened, written and closed again. Since the HDF5 open and close

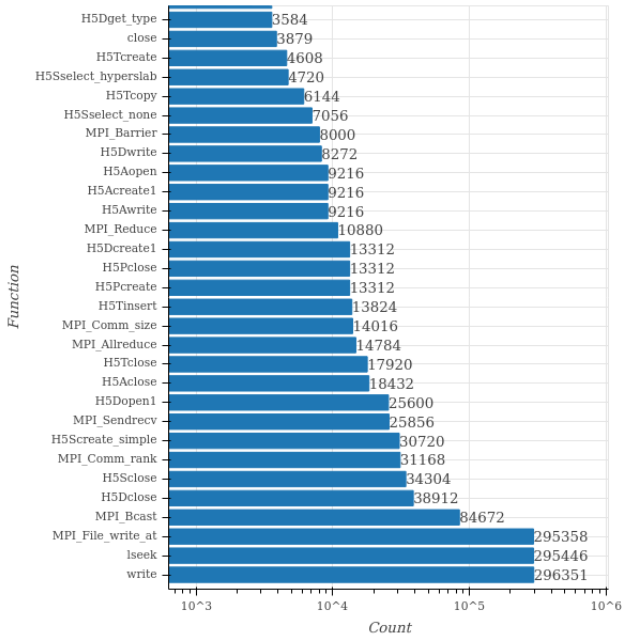


Figure 15: FLASH: the most visited functions counter

calls are both collective calls, this pattern may lead to performance issues especially when running at a large scale.

4.3 NWChem

We run NWChen using a water molecule sample input (h2o_scf.nw). During the computation, each rank writes to a local temporary file (named h2o_scf.junk.[RANK]), which is not a desired behaviour for large scale runs. For example, a million-cores run means creating and writing a million files within one directory at the same time, which is very stressful for parallel file systems.

Moreover, NWChem performs overlapping (actually conflicting) accesses to an runtime database file (h2o_scf.db) as shown in Figure 16. This file is only accessed by rank 0 but we observed both write-after-write, read-after-write and write-after-read patterns. Application users need to be careful when choosing parallel file systems because on file systems that support weak consistency models (e.g., BurstFS [7]), this conflicting pattern may cause consistency issues.

4.4 Chombo

Chombo has a similar issue as NWChem: it uses a file-per-process write pattern for each local output. Again, this could lead to scalability issues. Other than that, HDF5 is used to save the final result. The access patterns of each rank is shown in Figure 17. Each rank writes its local data to a shared file, and this pattern can be optimized by using collective I/O.

4.5 QMCPACK

For QMCPACK, we run a short diffusion Monte Carlo calculation of a water molecule. It runs on 16 ranks and writes out 5 checkpoint

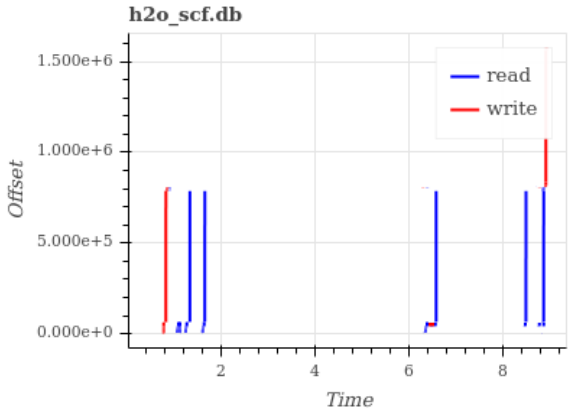


Figure 16: NWChem: conflicting patterns observed on file h2o_scf.db

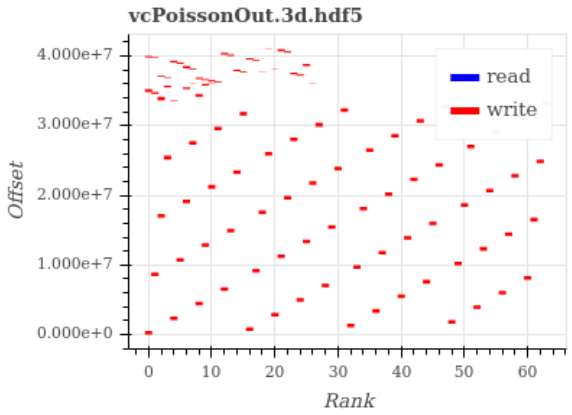


Figure 17: Chombo: HDF5 access pattern

files. Figure 18 depicts the access pattern of one checkpoint file over time.

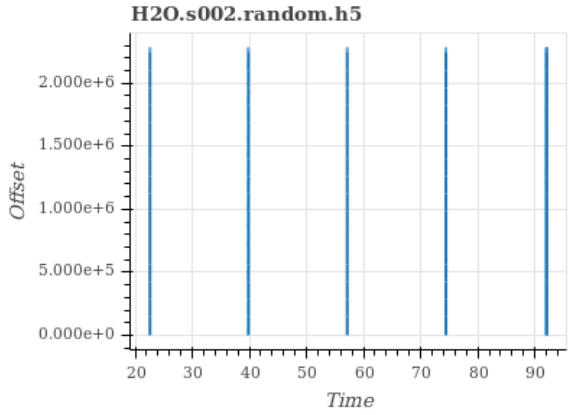


Figure 18: QMCPACK: overwrite the same checkpoint

Instead of writing to a new checkpoint file at every checkpoint step, QMCPACK overwrites the same checkpoint file. This is a dangerous pattern as failures may occur during the checkpoint step, and if an failure happens, the checkpoint file will be corrupted and the previous checkpoint will also be lost.

ACKNOWLEDGEMENT

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences under Award Number DE-AC05-00OR22725.

REFERENCES

[1] 2019. Flash Center for Computational Science. http://flash.uchicago.edu/site/flashcode

[2] Mark Adams, Peter O Schwartz, Hans Johansen, Phillip Colella, Terry J Ligocki, Dan Martin, ND Keen, Dan Graves, D Modiano, Brian Van Straalen, et al. 2015. *Chombo software package for amr applications-design document.* Technical Report.

[3] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 2009. 24/7 characterization of petascale I/O workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 1–10.

[4] Jeongnim Kim, Andrew D Baczewski, Todd D Beaudet, Anouar Benali, M Chandler Bennett, Mark A Berrill, Nick S Blunt, Edgar Josué Landinez Borda, Michele Casula, David M Ceperley, Simone Chiesa, Bryan K Clark, Raymond C Clay, Kris T Delaney, Mark Dewing, Kenneth P Esler, Hongxia Hao, Olle Heinonen, Paul R C Kent, Jaron T Krogel, Ilkka Kylänpää, Ying Wai Li, M Graham Lopez, Ye Luo, Fionn D Malone, Richard M Martin, Amrita Mathuriya, Jeremy McMinis, Cody A Melton, Lubos Mitas, Miguel A Morales, Eric Neuscamman, William D Parker, Sergio D Pineda Flores, Nichols A Romero, Brenda M Rubenstein, Jacqueline A R Shea, Hyeondeok Shin, Luke Shulenburger, Andreas F Tillack, Joshua P Townsend, Norm M Tubman, Brett Van Der Goetz, Jordan E Vincent, D ChangMo Yang, Yubo Yang, Shuai Zhang, and Luning Zhao. 2018. QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter* 30, 19 (apr 2018), 195901. https://doi.org/10.1088/1361-648x/aab9c3

[5] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus JJ Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L Windus, et al. 2010. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181, 9 (2010), 1477–1489.

[6] Chen Wang, Jinghan Sun, K Mohror, and E Gonsiorowski. 2020. Recorder 2.0: Efficient Parallel I/O Tracing and Analysis. In *The IEEE International Workshop on High-Performance Storage*.

[7] Teng Wang, W Yu, K Sato, A Moody, and K Mohror. 2016. *BurstFS: A Distributed Burst Buffer File System for Scientific Applications.* Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).