H5hut: A High-Performance I/O Library for Particle-based Simulations

Mark Howison*, Andreas Adelmann†, E. Wes Bethel*, Achim Gsell†, Benedikt Oswald†, Prabhat*

* Computational Research Division
Lawrence Berkeley National Laboratory
One Cyclotron Road, Berkeley, CA 94720, USA
Email: {mhowison,ewbethel,prabhat}@lbl.gov

† Accelerator Modeling and Advanced Simulations Group
Paul Scherrer Institut, CH-5234 Villigen, Switzerland
Email: {andreas.adelmann,achim.gsell,
benedikt.oswald}@psi.ch

***Abstract*—Particle-based simulations running on large high-performance computing systems over many time steps can generate an enormous amount of particle- and field-based data for post-processing and analysis. Achieving high-performance I/O for this data, effectively managing it on disk, and interfacing it with analysis and visualization tools can be challenging, especially for domain scientists who do not have I/O and data management expertise. We present the H5hut library, an implementation of several data models for particle-based simulations that encapsulates the complexity of HDF5 and is simple to use, yet does not compromise performance.**

***Index Terms*—Parallel I/O**

I. Introduction

Particle accelerators have enabled some of the most remarkable discoveries of the 20th century and are a cornerstone of research in fields ranging from basic science to applied biologics. Accelerator-based systems have now been proposed to address problems related to energy, biology, and the environment that are of great social importance. The design, optimization, and operation of these machines—some of the most complex ever built by mankind—requires both advanced numerical methods and high-performance computing (HPC) tools.

Particle-based simulations of accelerators, especially in six dimensional phase space, generate vast amounts of time-varying data. Reading and writing enormous datasets for post-simulation analysis remains challenging, especially on massively parallel high-performance computing systems. The HDF5 Utility Toolkit (H5hut) simplifies these I/O tasks by encapsulating specific data models in an easy-to-use C/C++ and Fortran API that has been tuned to perform and scale well on modern parallel file systems.

Although we focus on particle accelerator simulations in this paper, H5hut naturally accommodates any time-varying data that can be described using particles, rectilinear grids, or finite element meshes. H5hut has also been integrated with FastBit [1] indexing technology to accelerate queries of particle data and with analysis tools such as the ROOT data analysis framework and both the VisIt and ParaView parallel visualization tools.

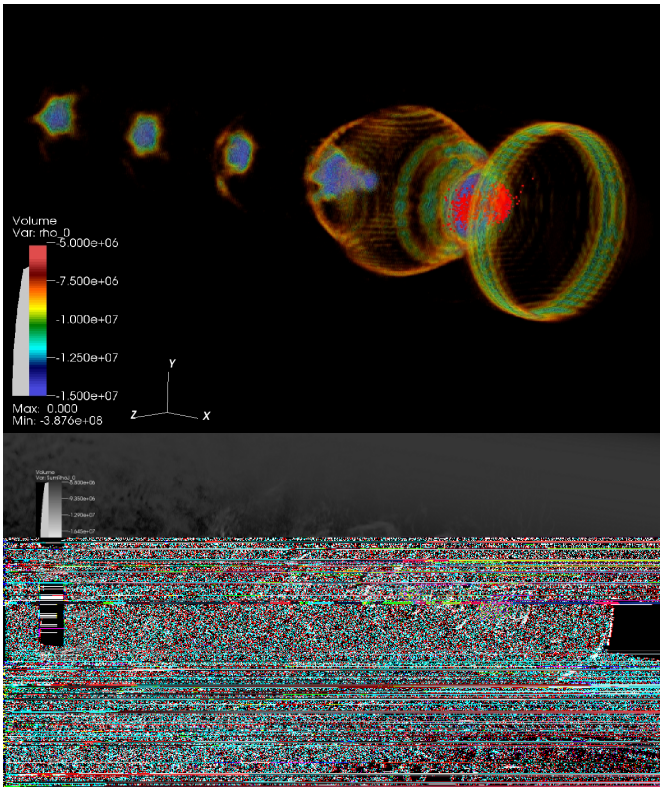

Fig. 1. These volume renderings of plasma density data from a laser wakefield particle accelerator simulation were visualized from H5hut output files using VisIt. Features of interest have been extracted and highlighted in the visualization using state-of-the-art indexing techniques.

II. Related Work

H5hut is built on top of the Hierarchical Data Format v5 (HDF5) file format and I/O library [2]. HDF5 provides several important features: it has a self-describing, machine-independent binary file format; it supports scalable parallel I/O for MPI codes on a variety of HPC systems; and it works equally well on workstations or laptop computers. HDF5's "object database" data model enables users to focus on high-level concepts of relationships between data objects rather than descending into the details of the specific layout of every byte

in the data file.

Another common I/O solution for scientific data is the Network Common Data Form (netCDF) library [3], which offers flexible data models and machine independence similar to HDF5. The most recent version, netCDF-4, has adopted HDF5 as its intermediate layer and inherits the parallel capabilities of HDF5. Prior to that release, the pNetCDF [4] library provided parallel support for the netCDF-3 file format. We chose to build on top of HDF5 rather than netCDF because of HDF5's richer set of tuning parameters for file layout and parallel I/O available through its "property list" interface.

The Geodesic Parallel IO (GIO) library [5] has similar goals for ease-of-use as H5hut, but targets a different science domain, namely climate modeling. It implements a data model for geodesic grids using both netCDF-4 and pNetCDF bindings.

The F5 library [6] builds on HDF5 to implement more complicated data models. It supports a range of grids, meshes, and fields by providing building blocks based on the concept of fiber bundles form algebraic geometry [7]. F5 implements user-defined and compound types in HDF5 to encapsulate geometric objects, like multi-vectors, and takes advantage of HDF5's support for type-casting, endian-conversion, and transformations on the layout of types. While F5 is intended to be modular and re-usable rather than to provide specific data models, it does provide an example usage for particle-based data models. However, F5 lacks the support that H5hut has for parallel I/O and for writing large datasets on modern HPC systems.

The Silo library [8] also provides data model abstractions on top of the HDF5 and NetCDF libraries. It can represent many mesh types, variable types, parallel decompositions, and advanced abstractions such as material volume fractions and species mass factions. In parallel usage, the Silo library uses a subset of processors to write out their own file, with a subsequent write to create a master file that contains links to those files. Although Silo can support the data models we are targeting, it has the same drawback as using HDF5 directly, which is that it provides unneeded features at the cost of additional complexity.

Both the Parallel Log-structured File System (PLFS) [9] and the Adaptable I/O System (ADIOS) [10] address the performance issues of parallel I/O on large HPC systems. PLFS is well-suited for check pointing, where the state of a running application needs to be quickly saved to disk in case of a system failure, and the files may never be read again. It uses file-per-processor writes to avoid the lock contention problems that arise with parallel access to shared files, but presents a virtual shared file by maintaining an index that maps the write calls from each process to global offsets. Although one could theoretically write an HDF5 or netCDF file into such a virtual shared file, it is unclear how that file would perform under read access. In PLFS, reads require the additional complexity of querying a global index file to lookup offsets into individual files in the log-structured container.

ADIOS supports both file-per-processor and collective access, and provides flexibility in how I/O is conducted through parameters specified in an external XML file that is read by the application at runtime. ADIOS also features performance optimizations such as asynchronous I/O, which double buffers data and offloads I/O operations onto designated I/O threads, allowing a computational code to continue non-I/O calculations while I/O is handled in the background. This optimization would also be possible in HDF5, although it has not been implemented yet. Although ADIOS provides interoperability with existing data formats like HDF5 and netCDF, a post-processing step is necessary to render such a file from the internal format used by ADIOS.

TABLE I
OVERVIEW OF H5HUT MODULES

Module	*Features*
Core	File and error handling, time steps, file and step attributes.
H5Part	Variable-length 1D arrays of particles.
H5Block	Rectilinear 3D scalar and vector fields, ghost zones, field attributes.
H5Fed	Adaptively refined tetrahedral and triangle meshes.

Unfortunately, neither PLFS nor ADIOS directly support the specific data models central to the particle-based simulations we are targeting. The goal of H5hut is to provide these data models through an easy-to-use API while also offering competitive parallel I/O performance.

III. DESIGN

H5hut is tuned for writing collectively from all processors to a single, shared file. Although collective I/O performance is typically (but not always) lower than that of file-per-processor, having a shared file simplifies scientific workflows in which simulation data needs to be analyzed or visualized. In this scenario, the file-per-processor approach leads to data management headaches because large collections of files are unwieldy to manage from a file system standpoint. On a parallel file system like Lustre, even the `ls` utility will break when presented with tens of thousands of files, and performance begins to degrade with this number of files because of contention at the metadata server. Often a post-processing step is necessary to refactor file-per-processor data into a format that is readable by the analysis tool. In contrast, H5hut files can be directly loaded in parallel by visualization tools like VisIt and ParaView.

H5hut is a veneer API for HDF5: H5hut files are also valid HDF5 files and are compatible with other HDF5-based interfaces and tools. For example, the `h5dump` tool that comes standard with HDF5 can export H5hut files to ASCII or XML for additional portability. H5hut also includes tools to convert H5hut data to the Visualization ToolKit (VTK) format and to generate scripts for the GNUplot data plotting tool.

A top design priority for H5hut was ease of use, especially for parallel I/O. The cost of HDF5's extensive functionality and flexibility is a complex API that can be daunting to inexperienced programmers and scientists, even for simple tasks such as writing out 1D arrays of data. Worse, effective parallel I/O is further complicated by the variety of data layout and tuning parameters available in HDF5. By restricting the

```
file = H5OpenFile("particles.h5", H5_O_WRONLY, MPI_COMM_WORLD);
H5SetStep(file, 0);
H5PartSetNumParticles(file, nparticles);
H5PartWriteDataFloat64(file, "x", x);
H5PartWriteDataFloat64(file, "y", y);
H5PartWriteDataFloat64(file, "z", z);
H5PartWriteDataFloat64(file, "px", px);
H5PartWriteDataFloat64(file, "py", py);
H5PartWriteDataFloat64(file, "pz", pz);
H5CloseFile(file);
```

```
fapl = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(fapl, MPI_COMM_WORLD, MPI_INFO_NULL);
file = H5Fcreate("particles.h5", H5F_ACC_TRUNC, H5P_DEFAULT, fapl);
step = H5Gcreate(file, "Step#0", H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
memspace = H5Screate_simple(1, nparticles, NULL);
MPI_Allreduce(&nparticles, &sum, 1, MPI_LONG_LONG, MPI_SUM, MPI_COMM_WORLD);
filespace = H5Screate_simple(1, sum, NULL);
MPI_Scan(&nparticles, &offset, 1, MPI_LONG_LONG, MPI_SUM, MPI_COMM_WORLD);
H5Sselect_hyperslab(filespace, H5S_SELECT_SET, &offset, NULL, &sum, NULL);
dxpl = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(dxpl, H5FD_MPIO_COLLECTIVE);
dset = H5Dcreate1(step, "x", H5T_NATIVE_DOUBLE, filespace, H5P_DEFAULT);
H5Dwrite(dset, H5T_NATIVE_DOUBLE, memspace, filespace, dxpl, x);
H5Dclose(dset);
dset = H5Dcreate1(step, "y", H5T_NATIVE_DOUBLE, filespace, H5P_DEFAULT);
H5Dwrite(dset, H5T_NATIVE_DOUBLE, memspace, filespace, dxpl, y);
H5Dclose(dset);
dset = H5Dcreate1(step, "z", H5T_NATIVE_DOUBLE, filespace, H5P_DEFAULT);
H5Dwrite(dset, H5T_NATIVE_DOUBLE, memspace, filespace, dxpl, z);
H5Dclose(dset);
dset = H5Dcreate1(step, "px", H5T_NATIVE_DOUBLE, filespace, H5P_DEFAULT);
H5Dwrite(dset, H5T_NATIVE_DOUBLE, memspace, filespace, dxpl, px);
H5Dclose(dset);
dset = H5Dcreate1(step, "py", H5T_NATIVE_DOUBLE, filespace, H5P_DEFAULT);
H5Dwrite(dset, H5T_NATIVE_DOUBLE, memspace, filespace, dxpl, py);
H5Dclose(dset);
dset = H5Dcreate1(step, "pz", H5T_NATIVE_DOUBLE, filespace, H5P_DEFAULT);
H5Dwrite(dset, H5T_NATIVE_DOUBLE, memspace, filespace, dxpl, pz);
H5Dclose(dset);
H5Pclose(dxpl);
H5Sclose(memspace);
H5Sclose(filespace);
H5Gclose(step);
H5Fclose(file);
H5Pclose(fapl);
```

Fig. 2. For writing out a typical particle array with six coordinates (position and momentum), H5hut (top) uses only 10 lines of code, while equivalent HDF5 calls (bottom) for implementing the same functionality and performance tunings require at least 35 lines.

usage scenario to particle-based simulations, H5hut encapsulates much of the complexity of HDF5 to present a simple interface, thus trading off some flexibility for ease-of-use. The code example in Figure 2 shows how 10 lines of H5hut calls can encapsulate the same functionality of 35 lines of HDF5 calls for writing a simple 1D array of particles. Also, we believe that H5hut's more verbose function names and fewer arguments per function (design choices that are shared by pNetCDF) produce code that is more readable for a domain scientist.

An H5hut file consists of a series of "time steps," which are HDF5 groups that are added sequentially to the file root. Each time step can hold multiple datasets, including 1D and 3D arrays and finite element data, of either 32- or 64-bit integer or floating point values. Attributes can be attached to the file or to an individual time step. See Table I for an overview of H5hut modules.

The H5Part module provides the data model for 1D arrays of particles. For writes, each MPI task can specify the number of particles it owns and its sequential "view" (in MPI-IO parlance) of the on-disk dataset is automatically calculated (as shown in Figure 2). For reads, a canonical view can be selected that evenly distributes the particles in a dataset across all tasks. In both cases, the user can also manually specify the start and end offsets of each task's view, or can specify a point selection using an array of offsets.

The H5Block module provides a data model for scalar and vector fields on rectilinear 3D grids. Again, a view

```
file = H5OpenFile("fields.h5",
  H5_O_WRONLY, MPI_COMM_WORLD);
H5SetStep(file, 0);
H5Block3dSetView(file,
  xrank*XDIM, (xrank+1)*XDIM - 1,
  yrank*YDIM, (yrank+1)*YDIM - 1,
  zrank*ZDIM, (zrank+1)*ZDIM - 1);
H5Block3dWriteScalarFieldFloat64(
  file, "Q", q);
H5Block3dWrite3dVectorFieldFloat64(
  file, "E", ex, ez, ey);
H5CloseFile(file);
```

Fig. 3. An example of H5Block calls for writing out a 3D grid.

```
file = H5OpenFile("mesh.h5",
  H5_O_WRONLY, MPI_COMM_SELF);
H5FedAddMesh(file, H5_TETRAHEDRAL_MESH);
H5FedBeginStoreVertices(file, nvertices);
int i;
for (i = 0; i < nvertices; i++) {
  H5FedStoreVertex(file, -1, Vertices[i].P);
}
H5FedEndStoreVertices(file);
H5FedBeginStoreElements(file, nelems);
for (i = 0; i < nelems; i++) {
  H5FedStoreElement(file, Elems[i].vids);
}
H5FedEndStoreElements(file);
H5FedAddLevel(file);
H5FedBeginRefineElements(file);
for (i = 0; i < nelems2refine; i++) {
  H5FedRefineElement(file, Elems2Refine[i]);
}
H5FedEndRefineElements(file);
H5FedCloseMesh(file);
H5CloseFile(file);
```

Fig. 4. An example of H5Fed calls for writing out a tetrahedral mesh wih one level of refined elements.

is used to represent which block of the grid is owned by each task. The example in Figure 3 shows how to create a view with a fixed block size of (`XDIM,YDIM,ZDIM`) for every task, and where each task has been assigned an index (`xrank,yrank,zrank`) into the grid. Ghost zones of arbitrary dimension can be represented as overlapping views, and H5Block features an algorithm to "reduce" these ghost zones so that all tasks' views are disjoint, thus eliminating redundant writes for the ghost zones.

The H5Fed module of H5hut provides a data model for adaptively refined tetrahedral and triangle meshes. Key features of H5Fed are tags, i.e. data associated with entities, and access to all up- and downward adjacencies. No intrinsic limits exist on the number of vertices, elements and level of refinements and multiple meshes can be stored in the same H5hut file. H5Fed is aggressively optimized for minimal memory and disk usage. Information that can be computed efficently from other data is neither stored on disk nor kept in memory. Currently, H5Fed only supports serial access, but a parallel version is in development.

H5hut is also designed to easily facilitate post-simulation analysis and visualization. H5PartROOT [11] is a tool to visualize medium scale data produced with H5hut and is based on the ROOT framework for data analysis developed at CERN. The main Graphical User Interface (GUI) allows convenient navigation between time steps and between several files for quick comparisons at the click of a mouse. Together with scripting capabilities, H5PartROOT covers almost all analysis task in particle accelerator design and optimization. The basic functionality of the tool ranges from plotting one-, two-, and three-dimensional particle distributions to line plots of step attributes such as emittance (projected, slice and screen), RMS beam size and centroid position, which are either read in directly from file or reconstructed from the particle distribution at the current time step. More sophisticated analysis tasks are possible using the corresponding ROOT classes.

Figure 1 shows an example of laser wakefield simulation data in H5hut format that has been rendered by VisIt. VisIt [12] is an open-source, high-performance, scalable visualization and analysis package for processing scientific data. It has been demonstrated to efficiently render terabytes of multi-variate scientific datasets on large HPC systems, and features plugins that directly import particle and field data in parallel from H5hut files.

IV. Applications

In the following two subsections, we illustrate two specific applications that write out as much as terabytes of particle and field data. In both cases, an I/O solution using H5hut was deployed using fewer lines of code than if the I/O routines had been written from scratch in HDF5. Additionally, the performance tuning we performed in the HDF5 and MPI-IO layers could be implemented once in H5hut to the benefit of both applications. In the final subsection, we describe how H5hut files can be augmented with bitmap indices to accelerate complex analysis tasks.

A. OPAL

OPAL (Object Oriented Parallel Accelerator Library) is a tool for simulating charged-particle optics in large accelerator structures and beam lines [13]. OPAL is based on IP^2L [14] and provides a data parallel approach for particles, fields and associated operators. OPAL is built from the ground up as a parallel application exemplifying the fact that HPC is the third leg of science, complementing theory and the experiment. This third leg is made possible now through the increasingly sophisticated mathematical models and evolving computer power available on the desktop and in scientific computing centers.

The data produced by OPAL are in general multivariate and describe a time-dependent, six-dimensional phase space. The time evolution of this space is governed by internal and external electromagnetic fields according to Maxwell's equations [15]. OPAL uses an FFT-based direct solver and a pre-conditioned conjugate gradient algorithm [16] to calculate the 3D space charge. Production runs of OPAL codes use

several thousands of cores, on the order of 10^9 simulation particles, and mesh resolutions on the order of 1024^3, which can be saved for post-run analysis using H5hut.

B. MC4

MC4 is a parallel particle-mesh based cold dark matter (CDM) code based on `MC2` [17] developed by Salman Habib (LANL), Katrin Heitmann (LANL), Robert Ryne (LBNL), and Viktor Decyk (UCLA). One of the goals of MC4 is to explore similarities between *beam dynamics* and *astrophysics* simulations and further develop IP^2L [14] in order to perform largest scale simulations in the two research areas.

MC4 makes use of the Friedmann - Lemaître - Robertson - Walker (FLRW) metric

$$ds^2 = c^2dt^2 - a(t)^2dx^2,$$

which is an exact solution of Einstein's field equations of general relativity. It describes a simply connected, homogeneous, isotropic expanding or contracting universe. The solution method splits the Hamiltonian into two pieces

$$H = H_{stream} + H_{gav}.$$

The gravitational component is solved as a Poisson problem with periodic boundary conditions, while the streaming component uses a second order Leap Frog integrator. MC4 is written entirely in C++ and makes use of the IP^2Lframework for parallel particle and field interactions. MC4 currently achieves scalable parallel performance for maximum values of $N_{particles} = N_{grid} = 4096^3$ using $O(10,000)$ cores, and uses H5hut to output several terabytes of particle and field data per time step. We are working toward problem sizes of $N_{partcles} = N_{grid} = 8192^3$ or even larger using $O(100,000)$ cores.

C. Laser Wakefield Analysis

Analysis and knowledge discovery from large, complex, multi-variate laser wakefield particle accelerator simulation data is a challenging task [18]. Researchers are interested in identifying beams of high-energy particles formed during the coarse of the simulations. While H5hut enables the output of large arrays of particle and field data from such simulations, efficient and accurate analysis of that data requires the ability to extract subsets that meet multi-dimensional range queries. For example, high-energy particles in laser wakefield data can be selected by thresholding for large momenta in the direction of the beam.

HDF5-FastQuery [19] is a high-level API that provides the ability to perform multi-dimensional indexing and searching on large H5Part datasets. It leverages an efficient bitmap indexing technology called "FastBit" [1] that is state-of-the-art in the database community. Bitmap indices are especially well suited for interactive exploration of large-scale read-only data. Storing the bitmap indices directly into the H5hut file significantly speeds up accessing subsets of multi-dimensional data and allows for portability of the indices across multiple computer platforms.

HDF5-FastQuery allows users to efficiently execute complex and compound range queries like

$$(energy > 10^5)\ \&\&\ (70 < pressure < 90)$$

and retrieve only the subset of elements in an H5Part dataset that meet the query conditions. Compared with other indexing schemes, compressed bitmap indices are compact and well suited for searching over multi-dimensional data even for arbitrarily complex combinations of range conditions.

V. Performance and Scalability

A key strategy for bringing single-shared-file performance up to the level of a file-per-processor approach is to employ "collective" optimizations, which have a long history of use in different MPI-IO implementations (see [20] for an overview). In general, collective optimizations use the additional information provided by a complete view of an I/O operation to decrease the number of I/O accesses and reduce latency.

By default, H5hut sets the MPI-IO "virtual file driver" in the parallel HDF5 layer to collective mode. This enables *collective buffering*, an optimization that assigns a subset of MPI tasks to act as "aggregators". Aggregators gather smaller, non-contiguous accesses into a larger, contiguous buffer in the first phase, and in the second phase write this buffer to the file system. On a system with a tuned collective buffering algorithm in the MPI-IO library, this can achieve bandwidths close to those of a file-per-processor approach. Most MPI-IO libraries use a heuristic to determine whether to enable collective buffering, but accept a "hint" to force collective buffering on. For instance, on a Cray XT system with a Lustre file system and version 3.2 or greater of the Message Passing Toolkit (MPT), setting the environment variable `MPICH_MPIIO_HINTS` to `"romio_cb_write=enable,romio_cb_read=enable"` will enable collective buffering.

We ran experiments on two machines, both with Lustre parallel file systems. JaguarPF is a Cray XT5 located at Oak Ridge National Laboratory with 672 Object Server Targets (OSTs). Franklin is a Cray XT4 located at the National Energy Research Scientific Computing Center with 48 OSTs.

Figure 5 shows the results of an experiment using the MC4 application that compares H5hut shared-file performance against a synthetic file-per-processor test that writes the same amount of data. This synthetic test was run using the IOR benchmarking utility [21]. Because the Lustre file system sets a hard limit of 160 OSTs over which a shared file can be striped, we also restricted IOR to use only 160 OSTs on JaguarPF (out of the 672 available).

We used IOR in POSIX mode, which means it can write a large amount of data into the OS write buffer, then return a bandwidth that is actually a measure of the memory bandwidth and not the I/O bandwith. To mitigate this effect, we modified IOR to allocate and touch a dummy array that filled 75% of available system memory. In previous experiments, we have found that this memory policy defeats the OS write cache during benchmarking. This step is necessary to accurately simulate HPC applications that use a significant portion of

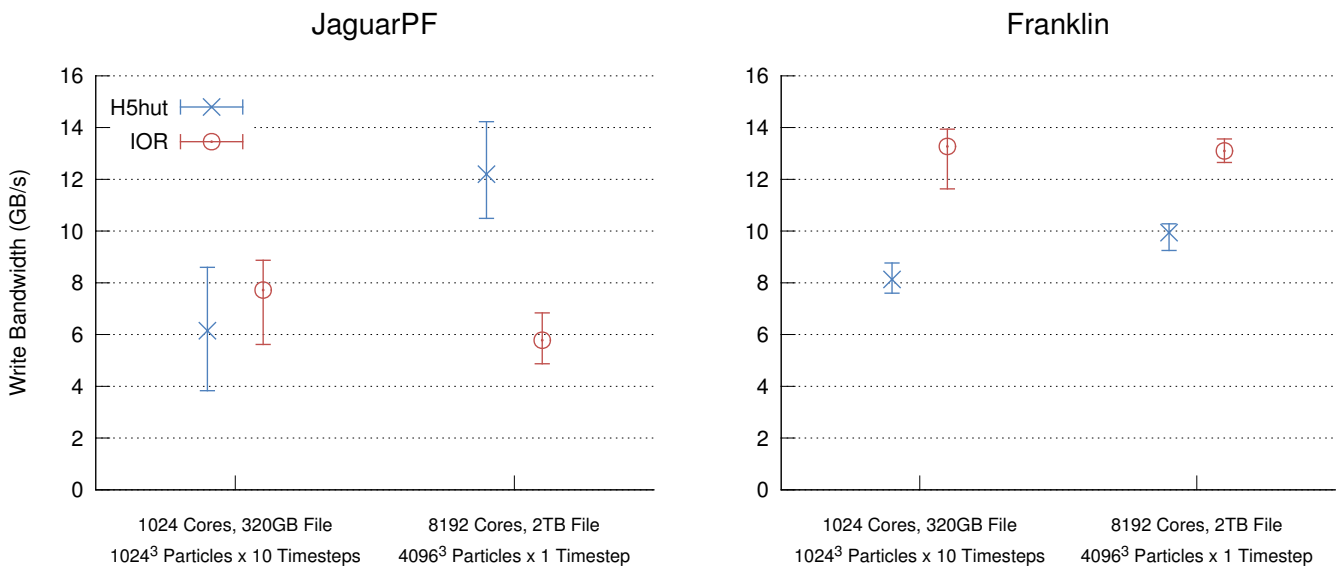

Fig. 5. A comparison of write bandwidths for H5hut output from the MC4 cosmology application and a simulated file-per-processor output using the IOR benchmark tool. Because of variability caused by contention with other users, we ran 4 to 7 trials and show the range and median value.

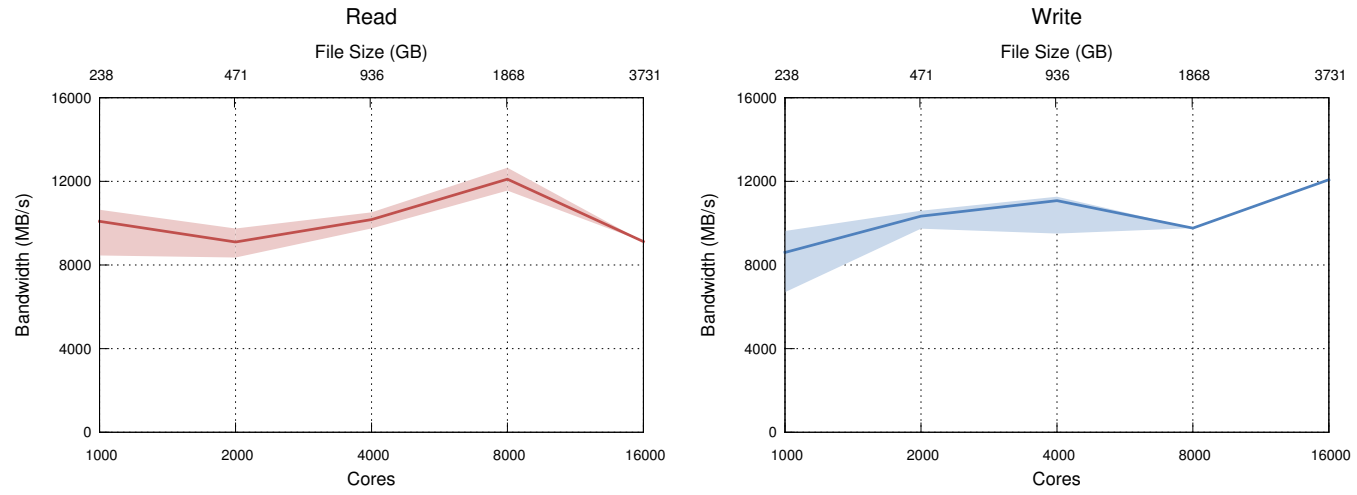

Fig. 6. Read and write bandwidths for a synthetic H5Block weak scaling study, scaling to 16,000 cores on Franklin and 3.7TB of data. The read times include a halo exchange, to transmit a ghost region of cells among neighboring blocks. The solid line shows the mean bandwidth, while the shaded region shows the minimum and maximum over repeated trials.

memory for their data and therefore would not leave memory available for OS write buffers.

For the write configurations we tested, file-per-processor performance dropped at higher core counts on JaguarPF, and H5hut's shared-file approach performed better. This result is corroborated by a study by Yu et al. [22], which found that file-per-processor performance lagged behind shared-file on JaguarPF when more than 96 OSTs were used. On Franklin, the file-per-processor approach maintained its performance out to 8,192 cores, but H5hut came within 75% of file-per-processor performance.

In another experiment (see Figure 6), we conducted a weak scaling study of the H5Block module up to 16,000-way concurrency and 3.7TB of data on Franklin. Because each processor writes the same size block, we are able to use a "chunked" layout in HDF5 (similarly called an "*N*-1 segmented" layout by Bent et al. [9]). A chunked dataset's elements are stored in equal-sized chunks within the file, allowing fast access to subsets of dataset elements, as well as the application of compression operations. HDF5 also allows the chunks to be padded out to a multiple of the Lustre stripe size by means of the "alignment" tuning property. When linked against the Lustre API, H5hut can automatically detect the stripe size of a file and set the HDF5 alignment property to this value.

We also bypassed the MPI-IO library by using a different "virtual file driver" (VFD) in the HDF5 layer called MPI-POSIX, which can be selected in H5hut with a flag at file open.

```
file = H5OpenFile("fields.h5",
  H5_O_WRONLY | H5_VFD_MPIPOSIX,
  MPI_COMM_WORLD);
H5Block3dSetChunk(file, XDIM, YDIM, ZDIM);
H5SetThrottle(file, 8);
```

Fig. 7. These H5hut calls enable the MPI-POSIX VFD, chunking, and throttling techniques used in the H5Block experiment.

The MPI-POSIX driver uses direct POSIX (e.g. `fwrite`) calls that are coordinated within the HDF5 library via MPI calls in a way that is analogous to the MPI-IO library operating in "independent" mode. In some scenarios, the lighter-weight MPI-POSIX driver exhibits better performance, especially on systems with poor MPI-IO collective performance, as was the case on the Cray XT prior to the release of MPT 3.2.

One problem with using independent rather than collective access to the file is that *all* MPI tasks, not just a subset, are communicating with the OSTs. With only 48 OSTs available on Franklin, 16,000 tasks caused time-outs in the Lustre client when writing so much data. To mitigate this, we introduced a "throttling" feature into H5hut that delays the write calls in a cyclic fashion by passing an MPI token. For example, setting the throttle factor to 8 for 16,000 tasks caused writes to be issued in batches of 2,000 tasks, which could complete before hitting the time-out limit. While it is unfortunate that the time-out problem is exposed at higher layers of the I/O software stack, we think it better to implement a workaround like throttling at the I/O library layer rather than to require applications to solve the problem independently.

Overall, our H5Block experiment showed near-peak performance on Franklin, which at the time of the experiment had 12GB/s peak read and write bandwidth as measured by daily IOR monitoring tests. The read phase also included a halo exchange (with a radius of one cell). At 16,000-way concurrency, the communication overhead of the halo exchange caused a drop in read performance. At 8,000-way concurrency, there was a dip in write performance, although we were only able to collect one data point and suspect that it suffered from contention with other users. Otherwise, we saw bandwidth increasing with concurrency, up to the 12GB/s peak.

VI. Conclusion

We have demonstrated how the H5hut library can be used to efficiently output terabytes of data from particle- and field-based simulations at up to 16,000-way concurrency on modern HPC systems. Moreover, implementing these I/O solutions with H5hut is easier and requires fewer lines of code than with HDF5 alone, and the resulting shared-file storage interfaces well with analysis and visualization tools.

Acknowledgment

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725; and resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] K. Wu, E. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Transactions on Database Systems*, vol. 31, pp. 1–38, 2006.

[2] The HDF Group, "Hierarchical data format version 5," 2000-2010, http://www.hdfgroup.org/HDF5.

[3] Unidata, "netCDF (network Common Data Form)," http://www.unidata.ucar.edu/software/netcdf.

[4] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003, p. 39.

[5] K. Schuchardt, "Data services for the Global Cloud Resolving Model (GCRM)," https://svn.pnl.gov/gcrm.

[6] W. Benger, "The Fiber Bundle HDF5 Library," http://www.fiberbundle.net/.

[7] W. Benger, A. Hamilton, M. Folk, Q. Koziol, S. Su, E. Schnetter, M. Ritter, and G. Ritter, "Using geometric algebra for navigation in Riemannian and hard disc space," in *Proceedings of Computer Graphics, Computer Vision and Mathematics*, Plzen, Czech Republic, 2008, pp. 80–92.

[8] Lawrence Livermore National Laboratory, "Silo: A mesh and field I/O library and scientific database," https://wci.llnl.gov/codes/silo.

[9] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: a checkpoint filesystem for parallel applications," in *SC '09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, Portland, Oregon, 2009.

[10] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, metadata rich IO methods for portable high performance IO," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, 2009.

[11] T. Schietinger, "H5PartROOT: a ROOT based graphical user interface for H5Part," 2006–2010, http://amas.web.psi.ch/tools/H5PartROOT/index.html.

[12] Lawrence Livermore National Laboratory, "Visit Visualization Tool," https://wci.llnl.gov/codes/visit.

[13] A. Adelmann et al., "The OPAL (Object Oriented Parallel Accelerator Library) Framework," Paul Scherrer Institut, Tech. Rep. PSI-PR-08-02, 2008–2010.

[14] A. Adelmann, "The IPPL (Independent Parallel Particle Layer) Framework ," Paul Scherrer Institut, Tech. Rep. PSI-PR-09-05, 2009.

[15] J. J. Yang, A. Adelmann, M. Humbel, M. Seidel, and T. J. Zhang, "Beam dynamics in high intensity cyclotrons including neighboring bunch effects: Model, implementation, and application," *Phys. Rev. ST Accel. Beams*, vol. 13, no. 6, p. 064201, Jun 2010.

[16] A. Adelmann, P. Arbenz, and Y. Ineichen, "A fast parallel Poisson solver on irregular domains applied to beam dynamics simulations," *Journal of Computational Physics*, vol. 229, no. 12, pp. 4554–4566, 2010.

[17] K. Heitmann, P. M. Ricker, M. S. Warren, and S. Habib, "Robustness of cosmological simulations i: Large scale structure," *Astrophys. J. Suppl.* **160**, *28*, 2005, [arXiv:astro-ph/0411795].

[18] O. Rübel, Prabhat, K. Wu, H. Childs, J. Meredith, C. G. R. Geddes, E. Cormier-Michel, S. Ahern, G. H. weber, P. Messmer, H. Hagen, B. Hamann, and E. W. Bethel, "High performance multivariate visual data exploration for extemely large data," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Austin, Texas, 2008, LBNL-716E.

[19] L. Gosink, J. Shalf, K. Stockinger, K. Wu, and E. W. Bethel, "HDF5-FastQuery: Accelerating complex queries on HDF datasets using fast bitmap indices," in *Proceedings of the 18th International Conference on Scientific and Statistical Database Management*, July 2006, LBNL-59602.

[20] R. Thakur, W. Gropp, and E. Lusk, "Optimizing noncontiguous accesses in MPI-IO," *Parallel Computing*, vol. 28, no. 1, pp. 83–105, 2002.

[21] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Austin, Texas, 2008.

[22] W. Yu, J. S. Vetter, and S. Oral, "Performance characterization and optimization of parallel I/O on the Cray XT," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2008.