bigfile : efficient and effective I/O of large and simple data

Extended Abstract

Yu Feng
University of California, Berkeley
Berkeley, CA
yfeng1@berkeley.edu

Rupert A. Croft
Carnegie Mellon University
Pittsburgh, PA
rcroft@cmu.edu

Tiziana Di Matteo
Carnegie Mellon University
tiziana@phys.cmu.edu

Markus Scheucher
Carnegie Mellon University
markus.scheucher@icloud.com

Ananth Tenneti
Carnegie Mellon University
tvsananth@gmail.com

François Lanusse
Carnegie Mellon University
flanusse@andrew.cmu.edu

Darren K Adams
University of Illinois at Urbana-Champaign
dadams@illinois.edu

Edward Karrels
University of Illinois at Urbana-Champaign
edk@illinois.edu

William Gropp
University of Illinois at Urbana-Champaign
wgropp@illinois.edu

Luu Huong
University of Illinois at Urbana-Champaign
luu1@illinois.edu

Jing Li
University of Illinois at Urbana-Champaign
lijing@illinois.edu

Simeon Bird
Johns Hopkins University
sbird@jhu.edu

ACM Reference format:
Yu Feng, Rupert A. Croft, Tiziana Di Matteo, Markus Scheucher, Ananth Tenneti, François Lanusse, Darren K Adams, Edward Karrels, William Gropp, Luu Huong, Jing Li, and Simeon Bird. 2017. bigfile : efficient and effective I/O of large and simple data. In *Proceedings of SC17, Denver CO USA, 2017,* 3 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

bigfile was originally developed as the I/O component of MP-Gadget which we used for BlueTides[1] cosmology simulation on NCSA BlueWaters[3]. at full machine scale (648,000 cores). Each checkpoint of our simulation contains 50 terabytes of data, saved in 420 seconds. We investigated and improved the performance of the library via the PAID program with the help of the parallel I/O team at NCSA. The library achieves the I/O bandwidth limit of BlueWaters file system at peak, while provides a variety of options to access the data and meta data after the simulation. We think as more HPC-like computing resources are made available to a variety of fields, bigfile can be useful in other HPC applications for efficient save and effective retrieval of table-like data and meta data.

The bigfile project is fully open source, hosted at https://github.com/rainwoodman/bigfile. The code base is stable, with occasional bug fixes. The core library consists of a few thousand lines of C code and can be fairly easily bundled into a large application or compiled as a library. The python interface is indexed on PyPI with binaries for Linux and OSX available on the BCCP conda channel.

In this poster, we will introduce the motivation, design, implementation, and the performance of the bigfile library and storage format. We welcome new users, comments, contributions, and inputs that may lead to a next new-feature version of the library.

[1] http://bluetides-project.org

SC17, 2017, Denver CO USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

Column	Length (N)	Data Type	Vector Size (M)
Position	N	Floating Point	3
Velocity	N	Floating Point	3
ID	N	Integer	1

Attribute	Data Type	Vector Size (M)
BoxSize	Floating Point	3
Time	Floating Point	1
Ω_M	Floating Point	1

Table 1: An example table-like columns and attributes produced by a 3-dimensional N-body simulation with N particles.

2 MOTIVATION

Computer simulations tend to produce large binary table-like data. The structure of these data is usually very simple. As an example, Table 1 lists the content of a snapshot from an N-body simulation typically used in the field of cosmology. Each column can be treated as an N by M table, where N is the number of entries, and M is the size of the vector, with $N \gg M$. In addition to the columns, we also need to store a few meta data entries, such as the time of the snapshot, the size of the simulation box, and parameters about the cosmology / background of the simulation.

As the size of simulation expands and as we employ increasingly larger parallel computing systems, it becomes a challenge to write these tables to the file system efficiently, utilizing full potential of the I/O subsystem. We also realize that data needs to be consumed and managed well after the initial production with very different tools. Therefore the data must be stored in a format that can be accessed easily by future applications on a variety of technology platforms.

HDF5 [8] allows one to easily access the table data of a computer simulation together with the associated metadata as attributes. However, native HDF5 does not support concurrent I/O within the same container object. Parallel HDF5 (HDF5-MPI) allows concurrent I/O, but requires extensive tuning / auto-tuning in order to achieve good performance.

An additional concern regarding a closed data storage format like HDF5 is that once the data goes into the container, a special library is needed whenever we want to retrieve the data. Although HDF5 group has made excellent progress in to simplify the access to the library e.g. (H5Lite), the interface is still inherently complicated due to the rich data model of HDF5. In many cases, researchers in our field prefers to first dump the HDF5 to ASCII or binary files before consuming the data.

Due to both concerns, in our field of cosmology simulation, a one file per rank/group model still dominates the field [see e.g. 5], which requires a special treatment at later retrieval to recombine the files, and majority of simulations uses simple binary data containers. An alternative solution seen in our field is to work with Lustre file system and increase the striping to create one giant physical binary file. [11]

3 OUR SOLUTION: BIGFILE

bigfile is our answer to the problem of simple data I/O in and beyond a parallel application. We describe our design choices in this section from the data producer side and the data consumer side.

3.1 Data producer: inside the parallel application

bigfile codifies the best practices for massively parallel I/O recommended by the BlueWaters team [1]: we create a moderate number of blob files for each column, then use a large but not excessive number of concurrent file system clients (Writer), evenly distributed across the entire application. We implement two modes to control the total number of concurrent file system clients: gang-throttling, where each MPI rank in a sub-MPI communicator takes turns to perform IO, and aggregation, where the root rank in a sub-MPI communicator aggregates the data within the sub communicator to perform IO. Both are well understood IO strategies, and as expected, the gang-throttling method is more efficient in the large data limit, while the aggregation method is more efficient in the small data limit. Please refer to section 3.3.

3.2 Data consumer: a plain container format

We illustrate the anatomy of a bigfile in Table 2. bigfile makes use of the file system hierarchy to represent the hierarchy of data. The terminal nodes are data columns, represented as file system directories containing two special plain text files (“header“ and “attr-v2“), and many binary blob files (six digit hexadecimal numbers). Please refer to the poster for a more elaborative example.

The “header“ file stores the numpy type descriptor [9] that is has been widely utilized in Python for describing the types of binary scientific data. We also store the vector size M and number of entries N_i in each binary blob file (to avoid querying the file system on the size of the blob files).

The “attr-v2“ file stores the attributes associated with a column. For each entry, we store the name, type descriptor, vector size, a hexadecimal representation of the raw bytes, and then a human readable representation as a comment.

The purpose of the design is to ensure that one can access both the data and the metadata directly by poking around in the file system hierarchy. In fact, some clients of our datasets (especially Fortran users) prefers to access the data this way instead of calling the C or Python routines provided by bigfile.

The Python binding of bigfile maintains a similar API to that of h5py. We follow the de-facto numpy slicing data retrieval protocol [7, 10] to support efficient partial retrieval of data from a bigfile container. The library intefaces nicely with dask[6], on which we build our post-simulation parallel data analysis software package nbodykit[4].

3.3 Improvements via the PAID program

PAID (Petascale Application Improvement Discovery) [2] is a program at National Center for Supercomputing Applications (NCSA) that brings science and engineering teams to work together to help create and implement application improvements technologies.

We investigated the performance of bigfile with the NCSA PAID program as part of the effort to understand and improve the I/O performance of the BlueTides simulation campaign. Our analysis with Darshan shows that bigfile can sustain 550 Gb/s IO throughput, nearing the limit of BlueWaters. The PAID program helped us to identify unnecessary file system metadata requests during the file creation phase. These requests are redundant on a POSIX file system such as the one on BlueWaters. Eliminating these operations to bring a 60% reduction of I/O time in the MP-Gadget application.

We implemented a benchmark tool for bigfile, “bigfile-iosim“ via the PAID program. Three per-rank loads were tested on BlueWaters., small, medium, and large (see Table 3). Figure 1 shows the wall clock time spent in each operation of bigfile. We do not tune the parameters, but rather fixed the number of blob files and the number of writers to a “reasonable“ choice, 1/4 of the total number of ranks.

The small load case is dominated by the file-system metadata requests, and is expected to scale poorly on a distributed POSIX file system. This is indeed observed. However, the poor scaling does not imply practical limitation, as we see that even with up to 131,072 ranks, each operation only takes 3 seconds. We also note that for this case the aggregated I/O mode brings about 50% reduction of wall clock time comparing to the gang-throttling mode.

The medium load case is picked to mimic an ordinary computer simulation with about 4M elements per rank. The wall clock time for read, write, and and blob creation remains flat for less than 8192 ranks, beyond which the time starts to increase linearly against the size of the problem, due to exhausting the I/O bandwidth of

BigFile Entity	HDF Entity	File System entity	Example
File	DataGroup	Directory	Snapshot_000
Column	Simple Dataset	Directory	Snapshot_000/Position/
Dataset	Structured Dataset		collection of columns (Python API)
Data	Data	Binary Files	Snapshot_000/Position/00000
			Snapshot_000/Position/00001 ...
Data Descriptor	Layout	Text File	Snapshot_000/Position/header
Attributes	Attributes	Text File	Snapshot_000/Position/attr-v2

Table 2: Anatomy of a bigfile container. We show the corresponding file system construct. We also list similar entities defined HDF.

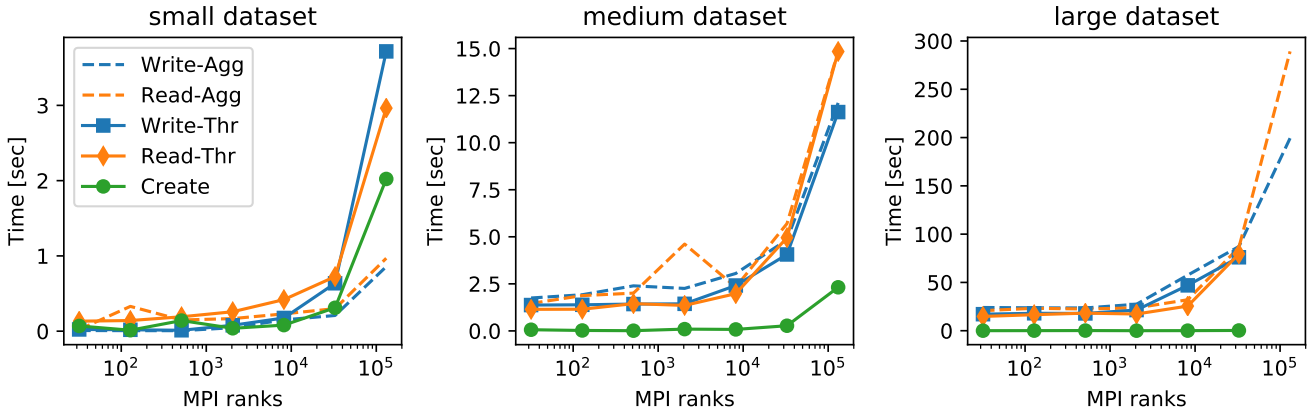

Figure 1: Benchmark of current bigfile implementation on BlueWaters

Test Name	*N* per rank	data type	vector size
small	4K	int32	1
medium	4M	int32	1
large	64M	int32	1

Table 3: Loads used in benchmarks.

the system. We also see that in this case the aggregated I/O mode is usually slower than the simple gang-throttling I/O mode, due to the extra time spent in migrating data during the aggregation. The large load case mimics a memory limited application (such as BlueTides simulation). The behavior is similar to the medium case, except the bandwidth limit kicks in at fewer ranks due to the heavier per-rank load.

4 CONCLUSIONS

We present bigfile, an effective and simple alternative solution to a simple problem. We welcome new users, comments, contributions, and inputs that may lead to a next new-feature version of the library.

ACKNOWLEDGMENTS

We acknowledge funding from NSF OCI-0749212, NSF AST-1009781 and the NCSA PAID program. The benchmarks are performed on NCSA BlueWaters computer. We thank the contributions from the external contributors of the bigfile project.

REFERENCES

[1] Andriy Kot. 2016. Parallel I/O Best Practices. (2016). https://bluewaters.ncsa.illinois.edu/documents/10157/169216/kot_io.pdf
[2] BlueWaters. 2017. PAID-IME. (2017). https://bluewaterrs.ncsa.illinois.edu/paid-ime
[3] Y. Feng, T. Di-Matteo, R. A. Croft, S. Bird, N. Battaglia, and S. Wilkins. 2016. The BlueTides simulation: first galaxies and reionization. *Monthly Notices of the Royal Astronomical Society* 455 (Jan. 2016), 2778–2791. https://doi.org/10.1093/mnras/stv2484 arXiv:1504.06619
[4] N. Hand and Y. Feng. 2017. A massively parallel toolkit for large scale structure. (2017). https://github.com/bccp/nbodykit
[5] D. Nelson, A. Pillepich, S. Genel, M. Vogelsberger, V. Springel, P. Torrey, V. Rodriguez-Gomez, D. Sijacki, G. F. Snyder, B. Griffen, F. Marinacci, L. Blecha, L. Sales, D. Xu, and L. Hernquist. 2015. The illustris simulation: Public data release. *Astronomy and Computing* 13 (Nov. 2015), 12–37. https://doi.org/10.1016/j.ascom.2015.09.003 arXiv:1504.00362
[6] The dask Project. 2017. Slicing. (2017). http://dask.pydata.org/en/latest/array-slicing.html
[7] The h5py Project. 2017. Reading and writing data. (2017). http://docs.h5py.org/en/latest/high/dataset.html#reading-writing-data
[8] The HDF Group. 2000-2010. Hierarchical data format version 5. (2000-2010). http://www.hdfgroup.org/HDF5
[9] The numpy Project. 2017. Data type objects. (2017). https://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html
[10] The numpy Project. 2017. Slicing. (2017). https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#basic-slicing-and-indexing
[11] M. S. Warren. 2013. 2HOT: An Improved Parallel Hashed Oct-Tree N-Body Algorithm for Cosmological Simulation. *ArXiv e-prints* (Oct. 2013). arXiv:astro-ph.IM/1310.4502