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

Yu Feng (Berkeley Center for Cosmology Physics & Berkeley Institute for Data Science, University of California Berkeley)
Markus Scheucher, Ananth Tenneti, François Lanusse, Rupert A. Croft, Tiziana Di Matteo (McWilliams Center for Cosmology, Carnegie Mellon University)
Darren K Adams, Edward Karrels, William Gropp, Luu Huong (University of Illinois Urbana Champaign)
Jing Li (National Center for Supercomputing Applications)
Simeon Bird (John Hopkins University)

Overview

We developed the bigfile library for storing data from cosmology simulations from HPC systems and beyond.
The library provides interfaces to store and retrive snapshots of BlueTides simulations run on NCSA BlueWater.
Each BlueTides snapshot contains 50 TB of table-like data for particle positions, velocities and more.

We present the library here to share it with the super-computing community.
We welcome new users, comments, contributions, and inputs that may lead to future enhancements to the library.

Two design goals

- Saving simulation snapshots to filesystem efficiently from a paralell ***MPI*** based application;
- Providing easy access to saved data from a variety of software platforms, esp from ***Python***;

Performance on HPC

The performance of the library was evaluated and improved through the NCSA PAID program. We find that it

- achieves the I/O bandwidth limit of the file system;
- requires little tuning for good performance;
- minimizes filesystem meta data requests;

The bigfile I/O solution is very simple, and it can be generalized for any parallel and large table-like data sets.
The bigfile library is fully open source, hosted at https://github.com/rainwoodman/bigfile.

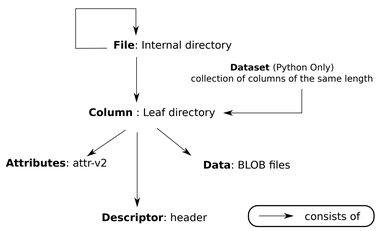
Codebase

- stable, with occasional bug fixes;
- core library consists of a few thousand lines of C code;
- bundled into a large application or compiled as a library;
- moderately covered by unit tests and coverity;

Data model

- bigfile does not aim to be a broad solution for all kinds of data; there is HDF5 for that.
- to simplify the model and code, we limit to the most useful usecase in simulations:
  very large, mostly immutable, table like binary data with small set of attributes;

Column: 2-D Table-like data, with shape (N, M), and a scalar type.
N is the number of entries (size), and M is the length of the vector.
N is predefined before writing. Streaming is not supported.
Dataset: multiple columns of the same N can be jointed as a Dataset.
File : A collection of columns of same or different sizes and files.
Attribute : a named single row vector, with a scalar type and length.



Comparison with HDF5

Data model of bigfile will look familiar because it is a subset of the data model of HDF5
However, the design of the data model of bigfile takes a different approach

- the HDF model is based on abstraction and representations of data ;
  parallel IO came as an after-thought (HDF-MPI).
- the bigfile model is based on parallel IO from MPI applications;
  a transparent container format is chosen to ease accessing data from a variety of software platforms

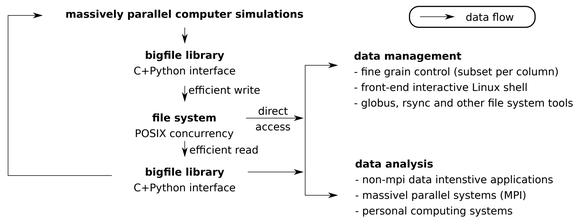
BigFile Entity	HDF Entity	File System entity	Example
File	DataGroup	Directory	Snapshot_000
Column	Simple Dataset	Directory	Snapshot_000/Position/
Dataset	Structured Dataset		collection of columns (Python API)
Data	Data	Binary Files	Snapshot_000/Position/00000 Snapshot_000/Position/00001 ...
Data Descriptor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

Using bigfile

Briging the gap

bigfile bridges the gap between MPI and non MPI applications;

- same protocol, different use cases.
- data production: MPI applications, parallel file systems, wallclock sensitive;
- data consumption: less wallclock sensitive; random access is important;
- data management : even less walllock sensitive;fine grain control is useful (e.g. partial data release)



bigfile API and Examples

File-like interface:

- open, close, seed, read, write, flush, set_attr, get_attr;
- easy to adapt to a large variety of applications;
- language bindings that have a native feeling;

Two API levels:

- Non-MPI: File and Column Creation, Random Read / Write,
  Attribute management
- MPI: Collective File and Column Creation, Collective Random Read / Write,
  Collective Attribute management

Language availability:

- Core implementation in C; C++ compatible;
- Python binding is based on the Non-MPI API, with its own MPI interface;
- Shell scripting via binutils and coreutils, and optionally bigfile tools
- Direct access in Fortran

Example: **Production of data in C**
- excerption from FastPM, libfastpmio/io.c

```
struct {double x[3], ....} * P; BigBlock bb; BigArray array; BigBlockPtr ptr;
big_file_mpi_create_block(&bf, &bb, "Position", "f4", 3,
                          Nfile, size, comm);
big_array_init(&array, &P[0].x[0], "f8", 2, (size_t[]) {NpLocal, 3}, NULL);
big_block_seek(&bb, &ptr, 0);
big_block_mpi_write(&bb, &ptr, &array, Nwriters, comm);
big_block_mpi_close(&bb, comm);
```

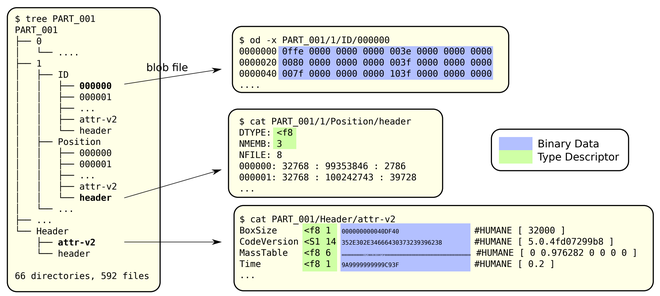
Example: **Consumption of data from Python**
- excerption from BlueTides data access guide

```
from bigfile import File, DataSet
f = File("PART_001")
d = DataSet(f, ['1/Position', 1/ID', '1/Velocity'])
print(d[:10])
```

Implementation

Anatomy of the Storage Container

- file system based hierarchy;
- only requires a functional file system for later data retrieval;
- reverse engineering the format is quick and fun;
- data is available even if bigfile is gone;



BLOB files

- On disk, C-contiguous, tight packed binary representation of data
- Evenly spliced into many physical files (avoid FS tuning)
- For large MPI jobs, a reasonable number will give good performance -- e.g. 1 per 4 ranks
- API provides random access
- Easy to open and stream

Descriptor (header)

- DTYPE and vector size (NMEMB)
- Number of BLOB Files and number of entries per file

Attributes (attr-v2)

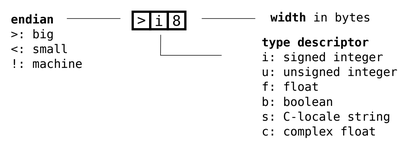
- One attribute per line
- DTYPE, vector size (NMEMB) and HEX encoding of binary data, followed by readable value

In-memory representation

- Strided dense array like FORTRAN / numpy: base pointer, strides, size, and dtype

DTYPE (data encoding)

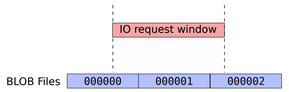
- borrowed From numpy's data type descriptor



Random Access

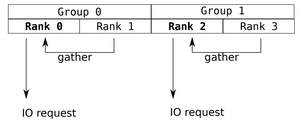
A column consists a few to thousands of BLOB files;
Random access is available for both MPI and non MPI applications
During random access, BLOB files are opened on demand:

- Avoids overflowing file handle limits of the operating system;
- Writing requires a correctly implemented POSIX non-overlapping;
- Reading requires less POSIX requirements;
  (Good because simulation data is nearly immutable)

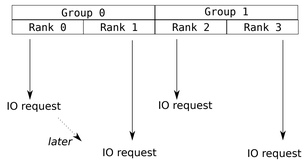


Parallel IO modes

Aggregated I/O:
- More data transfer on the network
- Less IO Requests



Gang Throttling I/O:
- Less data transfer on the network
- More IO Requests



Benchmark and Improvements via PAID

NCSA PAID program

PAID (Petascale Application Improvement Discovery) 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.

Benchmarks

Three benchmark cases; motivated by realistic scenarios:

- computing bound (small), moderate (medium), and memory-limit bound (large)

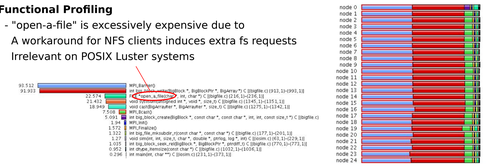
No fine-tuning: using 1/4 of ranks for concurrency and same number of BLOB files.

- 4 is the same as the default Lustre striping.

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

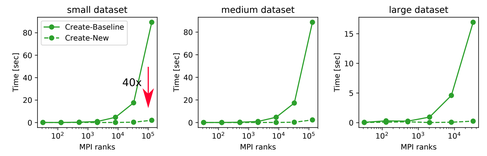
Functional Profiling

- "open-a-file" is excessively expensive due to
  A workaround for NFS clients induces extra fs requests
  Irrelevant on POSIX Luster systems



Optimized Column Creation

- Eliminating the NFS workaround improves the scaling; (New vs Baseline)
- up to 40x improvement at large scale;



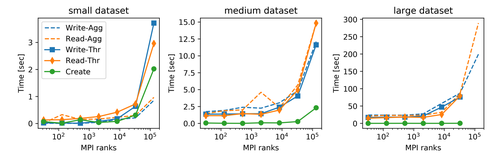
Wallclock by component

Pretty good performance without fine tuning

- reaching bandwidth limit at large scale
- getting acceptable performance for small dataset case (< 3 seconds)

We also see that throttling is usually faster than aggregation except the small dataset case
in which gang throttling is still pretty fast (~ 3 seconds at worst)

- gang-throttling transports less data on the network, but issues more file system IO requests
- aggregation transports more data on the network, but issues less file system IO requests



This work was supported in part by the BlueWaters PAID project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois.
We acknowledge funding from NSF OCI-0749212, NSF AST-1009781 and the NCSA PAID program. The benchmarks are performed on NCSA BlueWaters computer.