LA-UR-19-29393
Approved for public release; distribution is unlimited.

Title: Converting the MCNP Runtape to HDF5

Author(s): Josey, Colin James

Intended for: Report

Issued: 2019-09-18

Converting the MCNP Runtape to HDF5

Colin Josey

June 21, 2019

1 Introduction

The current runtape in MCNP is a Fortran unformatted IO dump of approximately 800 variables used for restarting an MCNP simulation. This format has a number of technical issues for those who interact with it. First, Fortran unformatted IO is not guaranteed to be portable. As a result, a runtape generated on one machine is not necessarily readable on another. Second, for users who wish to extract data from the runtape, there is no simple approach. It is impossible to extract one variable without knowing the size and position of every other one. Finally, the runtape itself is not backwards compatible such that a future version of MCNP could open it. Users are expected to keep around versions of MCNP that are compatible with saved runtapes.

For these reasons, there has been a great deal of interest in changing how the runtape is written to disk. Any changes would need the following properties:

- Individual variables in the runtape must be easily read in a variety of programming languages.
- The runtape must be portable from machine to machine and architecture to architecture.
- The resulting version of MCNP should not lose any features (specifically the multiple dump capabilities).
- The runtape must have backwards compatibility in mind.
- Performance should be on par or better than the current implementation.

The first two points are fulfilled by using the HDF5 file format. The remaining three components are discussed later on in this document. The current replacement meets all 5 requirements. This capability is planned for a future release of MCNP.

2 The F5 Library

Before the project began, a number of drawbacks to the HDF5 library needed to be addressed. While HDF5 is extremely versatile and capable, the API is very

verbose as a result. For MCNP, hundreds of variables would be written. Using this verbose syntax would lead to an immensely large and difficult to manage code base. Another problem is that the Fortran bindings need to be built with the same compiler version as is used to build MCNP itself. It is quite common to have HDF5 installed as a system library and use a variety of different compilers via module files to build the code during testing. In this circumstance, using the Fortran bindings would require multiple HDF5 installations. By using the C API, this can be sidestepped in the single-threaded case (in parallel HDF5, multiple versions would still be needed for each MPI installation). Finally, HDF5 requires users to manually close resources, which can easily lead to memory leaks and performance problems.

With these three drawbacks in mind, the F5 wrapper library was created. The main component is written in C++ which then links directly to the HDF5 C library. By leveraging shared pointers, whenever an HDF5 handle goes out of scope, the proper closing code can be called to free memory. And finally, a simple syntax was written to allow writing objects with a single line of code. On top of this, Fortran bindings were written to allow users to interface to the C++ component. As a result, Fortran code to write an array can be converted from this:

```
use HDF5

real(8)         :: foo(100)
integer(HID_T) :: file, space, dataset
integer         :: error

call h5fopen_f("file.h5", H5F_ACC_TRUNC_F, file, error)
call h5screate_simple_f(1, [100], space, error)
call h5dcreate_f(file, "dataset", H5T_IEEE_F64LE, space,
    dataset, error)
call h5dwrite_f(dataset, H5T_NATIVE_DOUBLE, foo, [100],
    error)
call h5dclose_f(dataset, error)
call h5sclose_f(space, error)
call h5fclose_f(file, error)
```

to this:

```
use F5_mod

real(8)       :: foo(100)
type(H5File) :: file

call file % open("file.h5", 'a')
call file % writeDataset("dataset", foo)
```

As a result, writing very large and complicated files can be done with ease.

One drawback to this simple syntax is a lack of control over the more advanced HDF5 features. Some of the more useful features, such as collective

MPI IO, compression, chunking, and compound types have been implemented in simplified forms. These advanced features are unused in the first attempt at rewriting the MCNP runtape, but were implemented for other projects and for future expansion.

Within MCNP, F5 is embedded in `dependencies/shacl-F5` as a git submodule. During building, the `shacl::cmake` toolset will automatically download F5 and its dependencies and add it to the build directory. This is a practical approach when one has network access to the F5 repository. For use beyond LANL, a script that automatically sets up a distributable source release with F5 included has been written.

One downside to the library is that the use of advanced Fortran features results in an increase in minimum compiler requirements. At the moment, GCC 7.4 and newer have been tested, as has Intel 18.0.1 and newer. GCC 6 has a known issue with Fortran finalizers that will result in continued memory leaks. As a result, MCNP's requirements will need to be incremented accordingly.

3 Runtape Design

At the moment, the HDF5 version of the runtape is very similar to the current MCNP runtape aside from the change in file format. The organization of the runtape is largely similar to the callgraph of `tpefil.F90`. As modernization continues, components of the runtape can be refactored into more sensible layouts. However, some initial reorganization has been performed. The next three sections will discuss some design decisions made in the design of the runtape.

3.1 Separation of Variable and Fixed Data

In the current runtape, there are several classes of variables: fixed-value-fixed-shape (`fixcom`), fixed-value-allocatable (`fdac`), variable-value-fixed-shape (`varcom`), variable-value-allocatable (`vdac`). At the beginning of simulation, the fixed value components are written once, to save space. The fixed-shape values are written first, followed by the allocatable ones (during read, the fixed shape values often contain array sizes necessary for allocation). Then, at each runtape dump, the variable values are written in similar fashion.

Due to the nature of HDF5, it is no longer necessary to separate the fixed-shape and allocatable objects, even if they are written at different times. This allows a three-group structure in the highest level of the runtape:

- `header` (containing information about the version of MCNP that wrote this runtape)
- `fixed` (containing values that are constant throughout the simulation)
- `variable` (containing the values that change throughout the simulation)

These groups are further subdivided. For example, `variable` contains numbered groups starting at '`1`'. Each corresponds to a different runtape dump. Within

these are currently unorganized variables that are written in `runtpw.F90` as well as further subgroups corresponding to features. The current layout of the runtape has been added to a document that is to be updated as the format is revised.

3.2 Multiple Dumps

One of the most complicated features in the runtape is the ability to store a finite amount of past dumps. As an example, the `PRDMP` card can be used to tell MCNP to keep the last 5 dumps and write every 5 batches during a k-eigenvalue simulation. In this example, on the 30th batch, the first dump is then removed from the runtape. The actual approach used to provide this capability is rather inefficient. Within `runtpq.F90`, the runtape is rewound and read to the start of the variable portion of the data. Then, two scratch files are opened. The current runtape is written to the first one. Then, each dump is read from the runtape (skipping those to be deleted) and re-written to the second scratch file. The runtape is rewound again, and the data in the second scratch is copied to the first. Finally, the first scratch runtape is copied over. As a result, if 5 dumps are to be kept and one removed, 10 dump writes are performed and 11 dump reads (one extra, as the runtape to be deleted is read to determine if it should be deleted) are performed for each dump. IO load can quickly get out of hand.

HDF5 allows for a far more efficient approach. At the start, the dump to be deleted is unlinked from the HDF5 file. This marks all of the disk space used by its contents as free for HDF5 to use. Then, the new dump is written in a new group (`variable/6/` and so on). HDF5 will write the new dump over the space reclaimed by the old one. A scalar indicating which dump is the newest is then incremented by one. If n is the number of dumps to be kept, this approach reduces IO from $4n + 1$ dump read/write operations to an unlink step and a single dump write.

3.3 Backwards Compatibility

As mentioned earlier, one of the goals of the project was to improve backwards compatibility. As a first attempt at implementing this feature, instead of checking a variable for the presence of a feature, the existence of the HDF5 group that contains said feature is checked. When the group exists in the runtape, data is loaded. If it does not exist, MCNP will load defaults. As a result, a runtape generated before a feature was added would simply load default settings for said feature.

Unfortunately, this can only help so much. Refactoring a data structure, for example, poses a particularly complex problem. One option would be to simply break backwards compatibility whenever such a change occurs, resulting in either frequent backwards compatibility breaks or a reluctance to perform necessary refactoring. A second option would be to version the data structures and provide loaders for multiple versions in MCNP. This would swiftly become intractably complex. The third option would be to provide conversion scripts

as data structures are transformed. These can be used by users to update the version of their runtapes. This third option is the most flexible and minimizes the code bloat in MCNP, but user support may become very difficult. At the moment, it is undecided which approach would be most practical.

4 Performance

While the change in performance from Fortran unformatted IO was expected to be relatively minor, it was still worthwhile to do some basic performance testing. For this, a Godiva sphere was overlaid by a 3D neutron flux mesh tally. The resolution of this mesh tally would be repeatedly increased by a factor of two in each dimension. It is tricky to measure the performance of just the IO portion of the run as there are numerous buffers between MCNP and the disk that prevent accurate timing. In order to test in spite of this, the entire simulation time was used[1]. Since transport slows down with very fine mesh tallies, only 10 particles were transported. Each simulation was run 10 times for the Fortran unformatted IO and for HDF5. The best simulation run time was kept. These runs were interleaved (one HDF5 run, then one Fortran IO run, etc.) to minimize the effect of external utilization of shared disk systems.

The results are shown in Figure 1 for a variety of different IO systems. XLan is shared NFS mount, Varan is a local solid-state disk, Scratch4 is a shared Lustre filesystem on a cluster, and Tmpfs is an in-memory filesystem on a single cluster node. For small mesh sizes, the overhead of creating the HDF5 file and the added complexity of the metadata HDF5 keeps results in a few percent performance loss. As the mesh size increases, the overhead becomes increasingly negligible. For some file systems, performance began to exceed the Fortran unformatted IO. For this problem, the smallest runtape (16x16x16) was 20.8 MiB, and the largest (1024x1024x1024) was 32.0 GiB.

5 Summary

This initial foray into rewriting MCNP's runtape in HDF5 has been largely successful in meeting the goals described in the introduction. Using HDF5, the runtapes are easily opened in a variety of other languages due to the many language bindings available. Similarly, HDF5 allows for portability between machines. No features were removed from MCNP during this conversion, and no significant performance regressions occurred. The one remaining challenge involves backwards compatibility. The initial groundwork has been laid, but there are a number of anticipated problems that will need to be solved as the project moves forward.

[1] It is possible to forcibly flush some of the buffers, but measuring the entire simulation time can be more reliable.

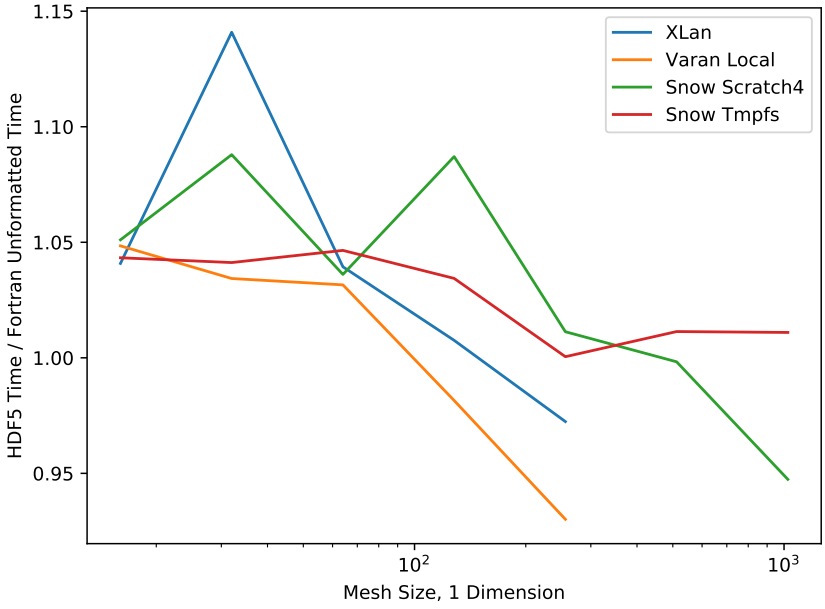

Figure 1: HDF5 vs. Unformatted IO Performance