Scalability Study and Performance Enhancement of a Monte Carlo Simulation Application Abeille on Perlmutter

Md Fuad Hasibul Hasan*
hasanm4@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York, USA

ABSTRACT

Monte Carlo simulation is a widely used technique for neutral particle transport simulations in general. With the advent of High-Performance Computing (HPC), new algorithms and techniques, it has become a widely used investigative study tool for application from fission or fusion reactor analysis, medical physics to a wide range of applications. For the former case, exceedingly large number of particles are simulated to get high fidelity results. Therefore, the scalability and performance on HPC systems are of utmost importance. Abeille is a newly developed open-source Monte Carlo simulation application that is designed to be more suitable for exploratory simulations to address the new challenges in Monte Carlo simulation. To use Abeille for large-scale simulations, it is important to study its scalability and performance. In this study, the strong and weak scalability of Abeille is studied and commented on. Then the bottlenecks are identified using multiple widely used profiling and tracing tools — namely TAU, Hpctoolkit as well as manual instrumentations. The major bottleneck is found to be the nuclear data library parsing. Nuclear reaction data is stored in thousands of individual files in a directory and each rank reads its own copy of data from the disk since Abeille needs each rank to have its own copy of data. To resolve this bottleneck, two different approaches are proposed and implemented: one, convert the nuclear data library to ADIOS2 or HDF5 format and read the data from that file using ADIOS2 and HighFive from each rank and two, read the data from a single rank and broadcast to all the ranks. The performance of these two approaches is compared and the results are presented. Both of the approaches were successful and was able to deliver at least 25 times to 50 times faster parsing rate (from over 25 seconds to less than 1 second.) The results also show that for less than 256 ranks, the first approach performs better, and for more ranks, the second approach should be adopted since it is founde to be well scaled for 256 ranks. However, this is dependent on the platform it is tested on. It is a major future scope of this study to implement parsing using any arbitrary number of ranks which will allow the users to fine-tune the performance based on the platform they are using.

CCS CONCEPTS

• **Applied computing** → **Physics**.

KEYWORDS

Monte Carlo Simulation, Scalability, Performance Enhancement, ADIOS2, HighFive

ACM Reference Format:
Md Fuad Hasibul Hasan. 2024. Scalability Study and Performance Enhancement of a Monte Carlo Simulation Application Abeille on Perlmutter. In . ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

With the advent of High-Performance Computing (HPC), Monte Carlo simulation has become a widely used technique for particle transport simulations. The applications of Monte Carlo simulation are diverse, ranging from fission and fusion reactor analysis to medical physics to proliferation studies[8]. This technique is particularly useful as gold standard for particle transport simulations due to its ability to provide accurate results with almost no approximations. However, the downside of Monte Carlo simulation is that it is computationally expensive, especially for large-scale simulations, e.g., fusion or fission reactor analysis. Newly developed variance reduction techniques have made Monte Carlo simulation more efficient. With the new emphasis on fusion energy, the need for efficient large scale Monte Carlo simulations — as a part of the multiphysics simulation — is more important than ever.

*This author solely conducted the study and wrote the report.

Project Report, April 23, 2024, Troy, NY

ACM ISBN 978-1-4503-XXXX-X/18/06

There are several widely used Monte Carlo simulation applications, such as MCNP, Serpent, and OpenMC. Abeille is a newly developed Monte Carlo simulation application that is designed to be more suitable for exploratory simulations to address the new challenges in Monte Carlo simulation. The special features of Abeille include the ability to use continuous material properties represenation, spacially continuous cross-sections, negative-weighted delta tracking, weight cancellation and neutron noise simulation. These makes it more suitable for exploratory simulations as well as for fusion reactor analysis[2].

To use Abeille for large-scale simulations, it is important to study its scalability and performance. Although, the author of Abeille has conducted some large scale simulations, a systematic study of its scalability and performance is missing. This project aims to fill this gap by conducting an extensive scalability and performance study of Abeille. After identifying the bottlenecks, several performance enhancement techniques are proposed and implemented.

This study is done on Perlmutter (a supercomputer at NERSC) with some degree of collaboration with author of Abeille, Dr. Belanger. He provided information and clarifications on different parts of Abeille implementations. Perlmutter is chosen because it will be used by the author of Abeille for large-scale multiphysics simulations later this year(2024).

This study is done in three phases. In the first phase, the strong and weak scalability of Abeille is studied. In the second phase, Abeille runs are profiled and traced to identify the bottlenecks and finally in the third phase, the parallel I/O performance is improved by using ADIOS2 and HighFive libraries[4, 5].

To understand the scalability of Abeille, it is important to understand the major components of Abeille. Abeille uses both shared and distributed memeory parallelism using OpenMP and MPI respectively. At the start of the simulation, the input fie containing the geometry and material specifications are read and the geometry is created which takes insignificant amount of time. Then the nuclear reacion properties are read from the nuclear data library using a thread from each rank and it takes a significant amount of time. This is followed by the simulation loop where the particles are transported through the geometry and the nuclear reactions are simulated. The total number of particles simulated is divided in equal parts among the ranks and each rank simulates its own particles in OpenMP threads. The simulation loop is the most time consuming part of the simulation. The simulation loops are scaling well if the number of particles simulated per thread is kept higher than a certain threshold. Finally, the result are written to the output file based on the user's specification. For these cases, tallies are not enabled due to the complexity of tallying process during the simulation.

The significance of this study is two fold. First, it will provide the author of Abeille with insights on the scalability and performance of Abeille and bottlenecks identified as well as bottleneck at reading cross-sections from disk solved. In simple words, it will be more adapted to leadership scale supercomputers. Second, the users of Abeille will be able to do large-scale simulations with Abeille more efficiently.

No GPU or CUDA acceleration is involved in this study since Abeille is not designed to use GPUs yet. Monte Carlo on GPU is still a diffcult problem, yet to be fully cracked due to Monte Carlo method's stochastic branching nature and random memory access patterns. Several groups are trying to implement GPU accelerated transport. We are also planning to implement GPU acceleration in near future.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the related works. Section 3 describes the methodology used in this study. Section 4 presents the results and discussion. Section 5 concludes the paper with a summary and future scope.

2 LITERATURE REVIEW & RELATED WORKS

One of the most closely related works to this study is the scaling study of OpenMC on Argonne's Blue Gene/P supercomputer. OpenMC is another widely used opensource Monte Carlo simulation application. First they did some preliminary studies to understand the expected performance in the light of Amdahl's law, thread overhead, load balancing, synchronized code in the threaded region, and memory bandwidth. Using 250 particles per thread, their weak scaling showed 65% efficiency for 48 cores. The performance was also satisfactory in absolute sense with speedups of 30 times on 48 cores. They found mainly two sources of performance degradation: scalability limitations due to shared resources in the memory subsystem were observed as cores were added and departures from ideal scalability were attributed to the memory subsystem's inability to deliver data to cores in a scalable manner [11, 14].

In a previous study, they also showed the strong scaling of OpenMC upto 2^{17} processors on Argonne's Blue Gene/P supercomputer. Their scalable parallel algorithm, especially the fission bank algorithm was the key to almost perfect strong scaling[12].

The abovementioned studies only focus on the particle transport simulation part of the Monte Carlo simulation. However, the performance of the nuclear data library reading part is not studied for both OpenMC and Abeille. Both OpenMC and Abeille read the nuclear data library from thousands of individual files in a directory and each rank reads its own copy of data from the disk [2, 12]. This is not a scalable approach since with the increase in number of ranks, the

disk I/O becomes exhausted and reading rate becomes stalled at a certain point.

For the case of Abeille, it uses a binary format of the ACE library which uses ASCII format to store the numbers. The ACE library is a widely used nuclear data library[3]. Prof. Belanger's thesis provides a detailed description of how the nuclear data library is read and used in Abeille[2]. In short, it uses a library called PapillonNDL (https://github.com/HunterBelanger/papillon-ndl) to convert the ACE library to a binary format as well as to read the data from the binary format during the simulation. To resolve the bottleneck at reading the nuclear data library, this PapillonNDL library is modified to read the data in parallel using MPI-IO with ADIOS2 and HighFive libraries.

Reading from a single file is more efficient than reading from a large number of files on disk due to performance benefits. Studies have shown that when accessing massive small files, reading from a single file significantly outperforms reading from multiple files. This is because reading multiple small files incurs high metadata access frequency and low disk efficiency, leading to lower performance compared to reading a single large file. Optimizations like extended read dir delegation and pre-read technologies between small files have been proven to reduce access latency and enhance overall performance significantly. Therefore, accessing data from a single file on disk is more efficient and faster than reading from numerous individual files, especially in scenarios involving small file access[6].

Therefore, to manage one large file creation and reading, the ADIOS2 and HighFive libraries are used. ADIOS 2 (The Adaptable Input Output System version 2) is an open-source framework designed to address scientific data management challenges, particularly in the context of scalable parallel I/O. The framework offers a unified application programming interface (API) focusing on n-dimensional Variables, Attributes, and Steps, abstracting the low-level details of data transport efficiently from application memory to various storage mediums. ADIOS2 also provides high-level APIs which can be seamlessly integrated with C++ (stream). Users can fine-tune runtime configuration parameters via XML and YAML files to optimize data movements without code recompilation. Additionally, ADIOS 2 supports data compression using third-party libraries for both lossy (e.g., zfp, SZ, MGARD) and lossless (e.g., blosc, bzip2, png) operations. All these features makes ADIOS2 a suitable candidate for Abeille to manage the nuclear data library reading[5].

HighFive is also a suitable candidate for Abeille to manage the nuclear data library reading becuase of its ease of use and it is already used in Abeille for writing the output files. It is a modern, header-only C++-friendly interface for libhdf5, providing seamless integration with HDF5 in C++ projects. It supports STL vector/string, Boost::UBLAS, Boost::Multi-array, and Xtensor, with automatic type mapping for C++ from/to HDF5. HighFive boasts a minimalist interface with no dependencies other than libhdf5, ensuring zero overhead and easy integration [4].

It is generally important to choose the simulation problem wisely, especially when it comes to performance studies. It allows to use these as reference points for comparison with other codes. Benchmark for evaluation and validation of reactor simulations (BEAVRS) is used as the simulation model for this study. BEAVRS is a multi-cycle full-core Pressurized Water Reactor (PWR) depletion benchmark that provides detailed reactor core models for testing and validation of coupled neutron transport, thermal-hydraulics, and fuel isotopic depletion. It includes information on fuel assemblies, burnable absorbers, in-core fission detectors, core loading patterns, and in-vessel components, enabling validation of coupled-physics codes and uncertainty quantification of in-core physics computational predictions[7]. There are several other benchmarks available for reactor simulations, such as VERA, CASL, etc. However, BEAVRS is chosen for this study because it is one of the most widely used benchmarks and it comes with the Abeille distribution.

3 PERFORMANCE AND SCALABILITY STUDY OF ABEILLE

In this section, the process of conducting the scalability study is described. TAU and Hpctoolkit are used to profile and trace the Abeille runs[1, 13]. For the description of the process, chonological order of decision making is followed.

One point to be clarified before the discussion is that when it is said that the simulation configuration is 1 node, it means that the number of rank is 2 and the number of threads per rank is 128. If the number of nodes is 2, the ranks and threads are doubled and so on. This configuration is chosen because Perlmutter has 2 sockets per node and 64 physical cores per socket and 128 virtual cores per socket. For all the executions, `--bind-to socket` option is used to bind the threads to the sockets[9]. Therefore, this confguration will keep the threads within the same socket to run on the same memory and minimize the MPI communication overhead.

First and foremost, the strong and weak scalability of Abeille is studied. There are two pieces of information are need for this: the time taken by Abeille and the number of particles simulated. Therefore, decision of the number of particles simulated is made first. The number of particles simulated has to be large enough to keep the threads busy for a significant amount of time but at the same time, it should be as small as possible to keep the simulation time within a reasonable limit to not burn the supercomputer's resources. After some inital runs, it is decided to simulate 250 particles

per thread. And there are 256 virtual cores per node on Perlmutter. It makes the total number of particles simulated to be $250 \times 256\times = 64000$ particles per node. At first, a primary decision was made only by observing how the threads are utilized using htop. However, to make the decision more scientific, 3 runs are conducted with 100, 150 and 250 particles per thread. The trace and profile analysis of these runs are done using Hpctoolkit and they show that for 250 particles per thread, the threads are utilized for a significant amount of time to do the `PowerIterator` operation.

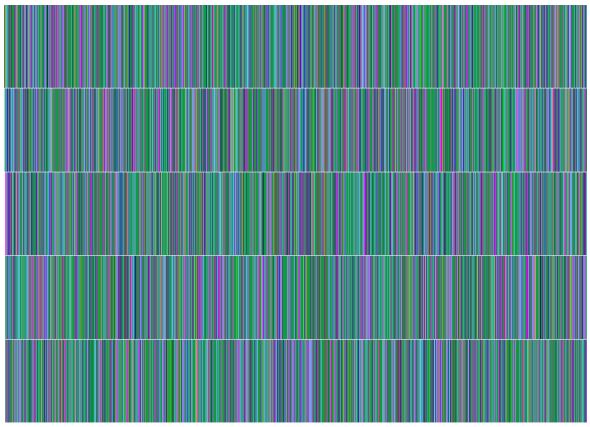

Figure 1: Trace of Abeille run with 100 particles per thread

Figure 1 shows the trace of Abeille run with 100 particles per thread. Each row represents a thread (only 5 threads among 256 threads are shown for brevity) and the x-axis represents the time. The color of the row represents the state (the function being executed) of the thread at that time. Colors in Hpctoolkit cannot be customized and it will be described in the writing what are the discussed colors are. Here, this trace shows that there is not enough work for the threads and they are mostly idle (the green color). A similar trend is observed for the run with 150 particles per thread. However, for the run with 250 particles per thread, the threads are utilized for a significant amount of time to do the `PowerIterator` operation.

Here in Figure 2, the trace of Abeille run with 250 particles per thread is shown. The blue color represents the `PowerIterator` operation and the pick vertical lines straight across the threads are the MPI communications.

This is also confirmed by the profile analysis done using TAU. The tracing graph showed in Figure 3 illustrates the zeroth and several other thread occupancy. It shows that all the nonzero threads are utilized for a significant amount of time and loads are balanced. It also shows the zeroth thread taking extra time to parse the nuclear data from the disk. Thais is discussed in the next section.

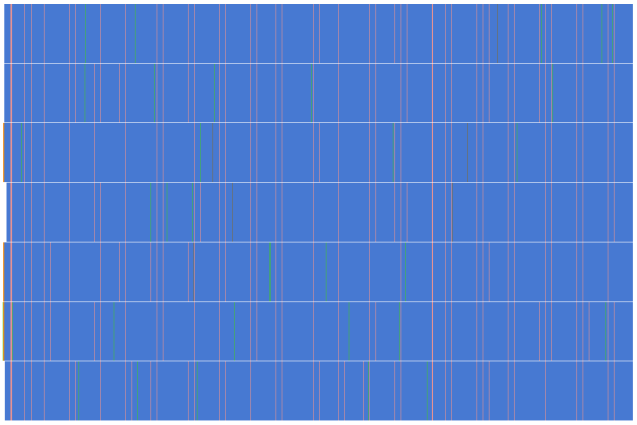

Figure 2: Trace of Abeille run with 250 particles per thread

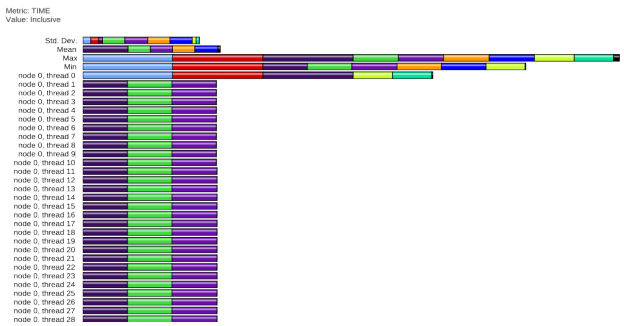

Figure 3: Traces of Abeille run with 250 particles per thread

3.1 Weak Scaling

After deciding on the number of particles simulated per thread to be 250, the weak scaling study is conducted. Starting from 1 node to 128 nodes — each node has 256 cores — which means the total number of particles simulated is $250\times256\times$number of nodes, keeping the number of particles simulated per thread constant. At first, the times reported by Abeille were used but later on, it was discovered that Abeille reports time from only one rank. This is not what is needed for weak scaling study becasue some rank may take longer time than others due to load imbalance or varius other reasons. For this reason, Abeille was manually instrumented to report time time taken by each sections of interests. There were several `MPI_Barrier` calls were introduced to make sure that the time reported is the time taken by all the ranks. To get the times, `std::chrono` library is used as follows.

```
MPI_Barrier(MPI_COMM_WORLD);
const auto start = chrono::steady_clock::now();
do_some_work(size);
const auto end = chrono::steady_clock::now();
```

Figure 4 shows the weak scaling. Since the number of particles per thread or node are not changing, the time should be constant along the x-axis. This plot shows two important phenomena. First, the total time is not scaling well after 16 nodes. Second, the responsibility of the divergenec from weak scaling mostly goes to the nuclear data library parsing section.

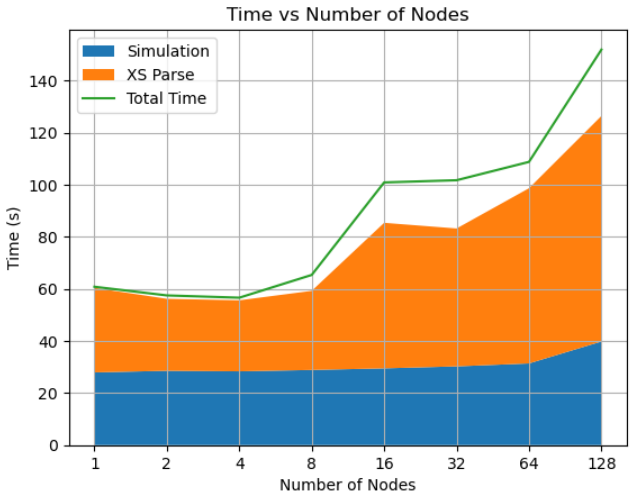

Figure 4: Weak scaling of Abeille

Therefore, it is clear that the major bottleneck is the nuclear data library parsing section. There are some other parts too, for example, the intilazation of the particle transport, source and source site sampling are also taking gradually increasing amount of time, especially after 16 nodes. However, for this project, I only focused on the nuclear data library parsing section. The implementation of the parallel I/O and the performance enhancement are described later in this section.

3.2 Strong Scaling

Strong scaling is also conducted in a similar way with the same number of particles per thread and noded as in the weak scaling study. The times are also extracted the same way. Figure 5 shows that it scales almost perfectly upto 64 or $16k$ cores. However, it is to be noted that this is only showing the time taken for the `PowerIterator` operation, not the total time taken by Abeille. Because the total time will not scale well because of nuclear data library parsing as shown in the weak scaling study.

There are several reason for divergenece from ideal strong scaling after 64 nodes. Mainly, the time taken by the MPI communications and collective operations during the `PowerIterator` operation is increasing due to the increase in the number of ranks.

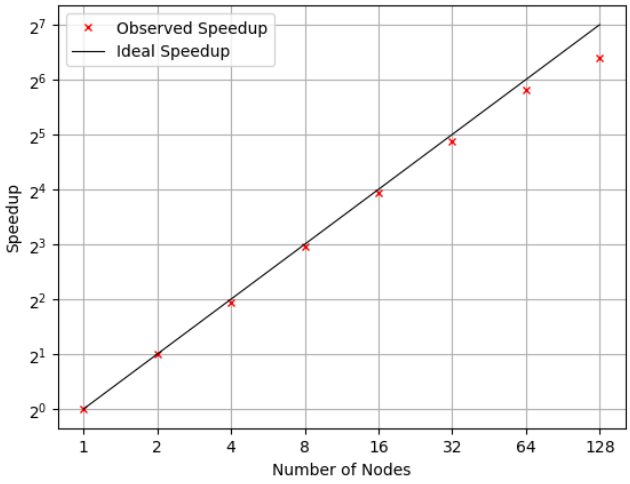

Figure 5: Strong scaling of Abeille

It is generally considered easier to achieve strong scaling than weak scaling for Monte Carlo simulation applications. Due to the nature of Monte Carlo simulation, the number of particles simulated per thread at each genearaton is simulated independently in parallel. In other words, This approach allows for a more straightforward distribution of computational load and resources, leading to improved efficiency and performance. Additionally, in simulations involving interfaces and domain wall pairs, weak scaling implies that certain properties, such as the contact distance between surfaces, exhibit a weak dependence on system size[10]. For Abeille's case, it showed relatively good strong and weak scaling upto 128 nodes or $32k$ processes if only the `PowerIterator` operation is considered. Therefore, it presents a strong case for using time and effort to improve the nuclear data library parsing.

3.3 Profiling and Tracing

In the above discussion, scaling results are only discussed based only on the theoretical point of view. However, to pinpoint the bottlenecks, the runs are profiled and traced using TAU and Hpctoolkit. At first, I used TAU to profile and trace. Interposition type instrumentation was used due to two reasons: first, no recompilation with TAU complier wrapper is needed (there is no working TAU compiler wrapper present now) and second, Abeille is a complex code which makes it extremely difficult to instrument manually with TAU. But the disavantage of this approach is that TAU wasn't able to capture the call stack which is needed to understand the bottlenecks. However, I later did several TAU profiled runs to understand the load balance and the time taken by MPI communications and collective operations. Figure 3 is already shown in the previous section. It also illustrates that

Table 1: Abeille Profile Summary

Scope	32 Node Time	1 Node Time
`parse_input_file`	63.15%	56.23%
`PowerIterator::run`	36.85%	43.77%

the zeroth thread is taking almost 50% of the time to parse the nuclear data from the disk marked by red. The other runs also showed similar results: zeroth thread taking much of it's time to parse the nuclear data and other ranks are well-balanced.

Figure 6: Time taken by each thread for 2 ranks

Moving on to the Hpctoolkit, load balancing can be illustrated more clearly with better insights. Figure 6 shows that the non zero threads are nicely balanced. And, the zeroth thread is taking long time. The slight imbalance actually comes from the fact that when it is said that there are 250 particles per thread, total number of particles are given to Abeille and it divides the particles among the threads — which can be slightly different for each thread based on the implementation of particle distribution.

Figure 7: Time taken by each thread for 64 ranks

For the case of 64 ranks, Figure 7 also shows nicely balanced threads. One extra set of threads are using less time than the remaining. After investigating the profile, if is found that these threads are allocated very few particles to simulate. This will be an important point to consider in the future scalability improvement of Abeille.

The profiles from Hpctoolkit also gave insights on the time taken by each major parts of Abeille. A summary of this is shown in Table 1. As expected, the `parse_input_file` fraction is taking more time for 32 node case which is already high.

Number of cycles for a function calls are also probed using Hpctoolkit. For power iterators, the cycle count was 6.96×10^{14} and 1.96×10^{13} in $3.28 \times 10^{5}s$ and $7.14 \times 10^{3}s$ for 32 and 1 node respectively. Therefore the scaling loss is:

$$\text{Scaling Loss} = 1 - \left(\frac{6.96 \times 10^{14}}{1.96 \times 10^{13}}\right) \times \left(\frac{7.14 \times 10^{3}}{3.28 \times 10^{5}}\right) \approx 22\% \quad (1)$$

4 NUCLEAR DATA LIBRARY PARSING PERFORMANCE ENHANCEMENT

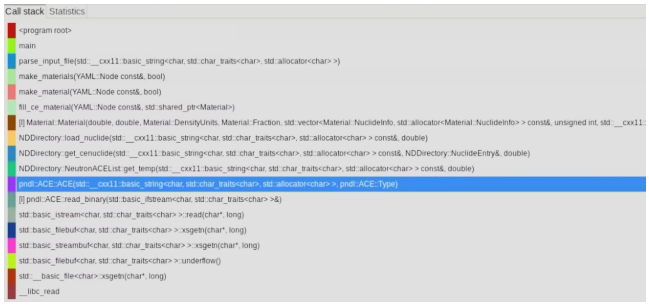

Figure 8: Call stack of ACE class

In the previous sections, it is established that improving the nuclear data library parsing scalability will improve the overall scalability of Abeille and it is more than half of the total time taken by the simulations. Two choices are made: one, to implement a new reading algorithm using ADIOS2 and HighFive from a single file and two, to experiment with reading the data from a single rank and broadcasting to all the ranks. The implementation process is described in detail in this section. I have followed the same order in the description as in the implementation.

The primary thing to understand is how the nuclear data library is loaded in Abeille. Using the Hpctoolkit's profiling and tracing, the call stack shows that to load each nuclide data, a simple class called `ACE` is used. It is part of the PapillonNDL library. The following Figure 8 shows the call stack. This is called deep into the `parse_input_file` function. Therefore, I needed to modify the `ACE` class to read the data from a single file using ADIOS2 and HighFive.

A nuclear data library contains thousands of individual files in a nested directory structure. For the case of ACE library, there are more than 7000 files for all the isotopes as well as separate files for different temperature conditions. In each file, there are some header information and data in ASCII format. The data is mainly consisted of some strings, doubles and most importantly, one big array of doubles. The `ACE` class represents and stores the data as follows:

```
ZAID zaid_;
double temperature_;
double awr_;
```

```
bool fissile_;
std::string fname_;
std::string zaid_txt;
std::string date_;
std::string comment_;
std::string mat_;

std::array<std::pair<int32_t, double>, 16> izaw_;
std::array<int32_t, 16> nxs_;
std::array<int32_t, 32> jxs_;
std::vector<double> xss_;
```

The ZAID object is just a pair of integers representing the atomic number and the isotope number. All the other variables are self-explanatory, at least, for the IO implementation. All the array and string sizes are fixed and known at the compile time except for the xss_ vector.

4.1 ADIOS2 Implementation

Therefore, the first step is to create a function that reads the the data from the given ACE library and stores it in a ADIOS2 supported data format. ADIOS2 supports several data formats, such as BP5, HDF5, BP4, etc. For this project, BP5 format is used. It provides significant advantages over its predecessors, particularly the BP4 format. It reduces memory consumption through Deferred Puts, which utilize user buffers for I/O, and optimized aggregation, leading to nearly half the memory consumption of BP4 in some cases. BP5 also improves metadata management, enhancing write/read performance for scenarios with hundreds of variables. Additionally, it provides improved functionality for appending multiple output steps into the same file, facilitating efficient restarts and enabling readers to filter out specific steps without corrupting existing data. While lacking burst buffer support for writing data, BP5's benefits make it a preferable choice for applications requiring efficient memory utilization, fast metadata operations, and streamlined file management.

In a single file, all the data is stored and each variable has it's own prefix based on the path it was stored in the ACE library. Because it will allow to use the same function calls to read the data from the file that uses the path to read and create the ACE objects. These are done in a member function of the ACE class calleed save_adios2. The implementation looks like the following. All of the the variables are not shown here for brevity.

Reading and then storing from this file efficiently is more challanging.

```
std::string gname = atom + "/" + id + "/";
io.DefineAttribute<double>(gname+"temperature",
              temperature_);
adios2::Variable<double> bpxss =
      io.DefineVariable<double>(gname+"xss",
          {xss_size}, {0}, {xss_size},
          adios2::ConstantDims);
bpWriter.Put(bpxss, xss_.data());
// save the other variables
bpWriter.PerformPuts();
```

The strings, integers and doubles are saved as attributes and the big array of doubles is saved as a variables following the convention of ADIOS2. Then, a function called ace2adios2 is created where all the ACE files from the ACE library are parsed using the ACE class and the data is saved in the ADIOS2 format.

ADIOS2 also provides MPI aware I/O operations and when enabled, it can decide the best number of threads for each rank to use to read the data from the file. It is very easy to enable. The user just needs to pass the MPI_Comm object while creating the adios2::ADIOS object, e.g., adios2::ADIOS adios(MPI_COMM_WORLD).

A large file is created in the output directory with the data in the ADIOS2 format. Another advantage of this format is that, after the conversion, the library size is reduced to 26 GB from 28 GB even without any compression.

Reading the data from this file and managing it efficiently is a more challanging task. If single ACE objects are read separately from this file, it will be very slow due to muliple file accesses. Therefore, only one adios2::IO object is created and the data is read in a single call. This is actually how it is done in Abeille or other Monte Carlo simulation applications. They create a list of materials from the input file and read the data from the ACE library. This allows ADIOS2 to read the nuclear data library asynchronusly.

By following the coding stylel of PapillonNDL, the ACE class is modified to create another constructor overload that takes the adios2::IO and adios2::Engine objects. This will allow to use the same IO and Engine objects to read all the ACE objects from the disk. The constructor signature looks like the following:

```
ACE::ACE(adios2::IO io, adios2::Engine bpReader,
              std::string atom, std::string id)
   : zaid_(0, 0),
     temperature_(),
     awr_(),
     fissile_(),
     fname_(),
     zaid_txt(10, ' '),
     date_(10, ' '),
     comment_(70, ' '),
     mat_(10, ' '),
     izaw_(),
     nxs_(),
     jxs_(),
     xss_() {
```

```
// read the data from adios2
read_adios2(io, bpReader, atom, id);
}
```

This constructor initializes the ACE object by reading the data from the ADIOS2 file. The `read_adios2` function is implemented as follows:

```
std::string gname = atom + "/" + id + "/";
// Read the variables
adios2::Variable<double> bpxss =
      io.InquireVariable<double>(gname+"xss");
temperature_ =
        io.InquireAttribute<double>(gname+
                    "temperature").Data()[0];
...
```

Then again, like writing the data, `readAdios2` function is created to read all the ACE objects from a list of materials and store it in `std::unordered_map` object to use it later in the simulation. Both serial and MPI aware versions of ADIOS2 were implemented and tested. The results are described later in this section.

4.2 HighFive Implementation

HighFive is also tested for the same purpose in search of the best performance. HighFive is a modern, header-only C++-friendly interface for libhdf5, providing seamless integration with HDF5 in C++ projects. The most important advantage is that implementing HighFive is very easy and it is already used in Abeille for writing the output files. The procedure for the implementation is exactly the same as ADIOS2. The only difference is the simpler API calls. It needed another constructor overload for the `ACE` class to read the data from the HDF5 file. The initializations are the same, only the function signature is different.

```
ACE::ACE(HighFive::File& hdf5_file,
                std::string gname);
```

The grouping feature of HDF5 is used to store the data in the file. The created `h5` format nuclear data library also has the same size of 26 GB. Reading and writing with HighFive is similar. Here is an example of reading the data from the HDF5 file:

```
HighFive::DataSet xss =
        hdf5_file.getDataSet(gname+"/xss");
xss.read(xss_);
```

From the perspective of parallel I/O, HighFive is supports collective I/O operations. But this is not useful for Abeille since it needs the nuclear data loaded in the memory of each rank.

4.3 Reading from a Single Rank

After analysing the performance of the ADIOS2 and HighFive implementations, it is seen that with the increase in MPI ranks, the time taken to read the data from the disk is increasing due to the saturation of the disk I/O. The algorithm is simple: instead of reading the data from the disk from all the ranks, only one rank reads the data and broadcasts to all the ranks. Therefore, there was no need to change the `ACE` class. But the ADIOS2 and HighFive formats are still used because of their faster reading speed even on a single rank. The algorithm is implemented as follows:

```
...
{initialize the adios, IO, Engine objects}
unordered_map<std::string, pndl::ACE> aceMap;
if (rank == 0) {
  ace2adios2(aceMap);
  store the data in the map
  serialialize the ACE object
  Broadcast the serialized data
}
else {
  receive the serialized data
  deserialize the data
}
...
```

Implementation of the `serialize` and `deserialize` functions are crucial for the performance. After several experimentation with different formats of the serialized data — namely, binary, iostream, vector of char — the best performance with reasonable complexity to handle the data is achieved with vector of char. The `serialize` function is implemented as follows:

```
void serialize(vector<char>& serialized_data) {
std::size_t total_size = zaid_txt.size() +  ...
serialized_data.reserve(total_size);
std::copy(zaid_txt.begin(), zaid_txt.end(),
          std::back_inserter(serialized_data));
for (std::size_t i = 0; i < xss_.size(); i++) {
  std::copy(reinterpret_cast<char*>(&xss_[i]),
    reinterpret_cast<char*>(&xss_[i])+sizeof(double),
      std::back_inserter(serialized_data));
}
... // serialize the other variables
}
```

The `deserialize` function is implemented the same way but in reverse. All the `vector<char>` objects are passsed by reference to avoid copying the data since the size of the data is very large — size of the `xss_` vector can about several hundred thousands of doubles.

4.4 Performance Results

The experimentation is done in the same way as Abeille reads the nuclear data library. For all the simulations presented in this study uses BEAVRS benchmark and it needs 200 different ACE objects to be loaded. When experimenting with the ADIOS2 and HighFive implementations, the same number of ACE objects are loaded using the same number of threads per rank — namely 2.

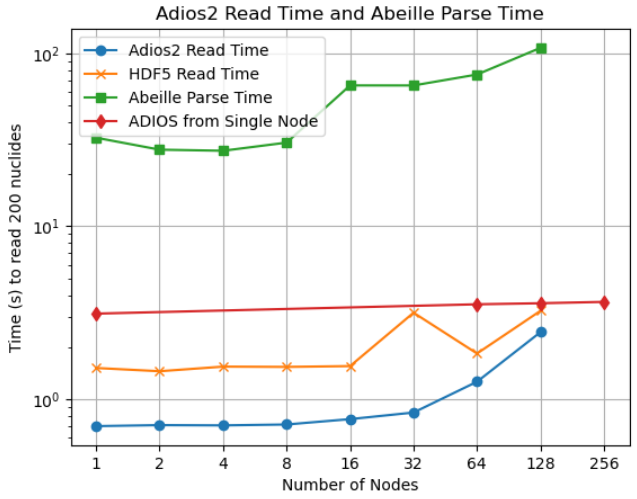

Figure 9: Performance comparison of ADIOS2 and HighFive

Figure 9 shows the performance comparison of ADIOS2, HighFive and reading from a single rank algorithm in comparison with the original PapillonNDL library. First of all, the ADIOS2 implementation is giving the best performance and scalability. Upto 32 nodes or 64 ranks, the time taken to read the data is almost constant and less than 1 second whereas the original PapillonNDL library takes more than 25 seconds. After 64 nodes, the time starts to rise quite rapidly and corsses 1 second. It can be considered as a significant improvement over the original PapillonNDL library.

The HighFive implementation is also giving a good performance. Although it is not as good as the ADIOS2, it is still taking less than 2 seconds to read the data from the disk. The jump seen at 32 nodes is still unknown and needs further investigation.

It can be easily noticed that the ADIOS2 and HighFive implementations again exploding after 128 nodes or 256 ranks. This is due to the saturation of the disk I/O. For this reason, the reading from a single or multiple user defined number of ranks is important. For now, the reading from a single rank is giving the best scalability and performance-wise, it is also reading with 4 seconds for upto 256 ranks.

5 CONCLUSION AND FUTURE SCOPE

In this study, the performance and scalability of Abeille is studied. The weak and strong scaling of Abeille is studied upto 128 nodes ($32k$ processes) on Perlmutter. It has shown several promising results, especially the scaling of the `PowerIterator` operation — which is the major part of Abeille and most difficult to scale — from both strong and weak scaling point of view. One significant bottleneck was identified: the nuclear data library parsing. The performance deteoration after 16 nodes is mainly due to this part. Therefore, the problem is addressed and solved using the ADIOS2 library and MPI broarcasting. The performance results are also promising. The ADIOS2 implementation is giving the best performance and scalability with the MPI communication.

This study has also opened several new research directions:

- Going further and experimenting with Abeille on even larger number of nodes to see how the nuclear data reading scales.
- Abeille should also be tested for more number of ranks to find out the bottlenecks in the `PowerIterator` operation which are not fully visible yet. And solve them.
- The performance of the ADIOS2 and HighFive implementations can be further improved by using the compression feature of these libraries as well as reading from specific number of ranks (may be user defined) and find out the best configuration.

ACKNOWLEDGMENTS

My sincerest thanks to Prof. Jacob Merson and Prof. Hunter Belanger for their guidance and support.

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] Hunter Belanger. 2022. *Development of numerical methods for neutronics in continuous media.* Theses. Université Paris-Saclay. https://theses.hal.science/tel-04213597

[3] Jeremy Lloyd Conlin. 2017. Listing of Available ACE Data Tables. (1 2017). https://doi.org/10.2172/1342828

[4] Adrien Devresse, Nicolas Cornu, Luc Grosheintz-Laval, Omar Awile, Tom de Geus, Fernando Pereira, Matthias Wolf, and HighFive Contributors. 2024. *HighFive - Header-only C++ HDF5 interface.* https://doi.org/10.5281/zenodo.10679738

[5] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. 2020. Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX* 12 (2020), 100561.

[6] Yang Hongzhang, Junwei Zhang, Xiangchao Zeng, Huanqing Dong, and Lu Xu. 2015. Research of Massive Small Files Reading Optimization Based on Parallel Network File System. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 204–212. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.97

[7] Nicholas Horelik, Bryan Herman, Benoit Forget, and Kord Smith. 2013. Benchmark for evaluation and validation of reactor simulations (BEAVRS), v1. 0.1. In *Proc. Int. Conf. Mathematics and Computational Methods Applied to Nuc. Sci. & Eng*, Vol. 7. 63–68.

[8] Dirk P. Kroese, Tim Brereton, Thomas Taimre, and Zdravko I. Botev. 2014. Why the Monte Carlo method is so important today. *WIREs Computational Statistics* 6, 6 (2014), 386–392. https://doi.org/10.1002/wics.1314 arXiv:https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.1314

[9] National Energy Research Scientific Computing Center (NERSC). [n. d.]. Perlmutter Architecture. https://docs.nersc.gov/systems/perlmutter/architecture/. Accessed: April 23, 2024.

[10] Paul Romano, Benoit Forget, and Forrest BROWN. 2011. Towards Scalable Parallelism in Monte Carlo Particle Transport Codes Using Remote Memory Access. *Progress in Nuclear Science and Technology* 2 (10 2011). https://doi.org/10.15669/pnst.2.670

[11] Paul K. Romano and Benoit Forget. 2013. The OpenMC Monte Carlo particle transport code. *Annals of Nuclear Energy* 51 (2013), 274–281. https://doi.org/10.1016/j.anucene.2012.06.040

[12] Paul K. Romano, Nicholas E. Horelik, Bryan R. Herman, Adam G. Nelson, Benoit Forget, and Kord Smith. 2015. OpenMC: A state-of-the-art Monte Carlo code for research and development. *Annals of Nuclear Energy* 82 (2015), 90–97. https://doi.org/10.1016/j.anucene.2014.07.048 Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013, SNA + MC 2013. Pluri- and Trans-disciplinarity, Towards New Modeling and Numerical Simulation Paradigms.

[13] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.

[14] Andrew R Siegel, Kord Smith, Paul K Romano, Benoit Forget, and Kyle G Felker. 2014. Multi-core performance studies of a Monte Carlo neutron transport code. *The International Journal of High Performance Computing Applications* 28, 1 (2014), 87–96. https://doi.org/10.1177/1094342013492179 arXiv:https://doi.org/10.1177/1094342013492179

Received 23 April 2024