Benefits of Parallel I/O in Ab Initio Nuclear Physics Calculations

Nikhil Laghave[1], Masha Sosonkina[1], Pieter Maris[2], and James P. Vary[2]

[1] Ames Laboratory/DOE, Iowa State University, Ames, IA 50011, USA
{nikhill,masha}@scl.ameslab.gov
[2] Physics Department, Iowa State University, Ames, IA 50011, USA
{pmaris,jvary}@iastate.edu

Abstract. Many modern scientific applications rely on highly parallel calculations, which scale to 10's of thousands processors. However, most applications do not concentrate on parallelizing input/output operations. In particular, sequential I/O has been identified as a bottleneck for the highly scalable *MFDn (Many Fermion Dynamics for nuclear structure)* code performing *ab initio* nuclear structure calculations. In this paper, we develop interfaces and parallel I/O procedures to use a well-known parallel I/O library in MFDn. As a result, we gain efficient input/output of large datasets along with their portability and ease of use in the downstream processing.

Keywords: Parallel I/O, HDF5, MFDn.

1 Introduction

The direct solution of the quantum many-body problem transcends several areas of physics and chemistry. Our aim is to solve for the structure of light nuclei addressing the hurdles of a very strong nucleon-nucleon interaction, three-nucleon interactions, and complicated collective motion dynamics, by direct diagonalization of the nuclear many-body Hamiltonian in a harmonic oscillator single-particle basis.

The main tool used for the study of nuclear structure is the software package MFDn (Many Fermion Dynamics for nuclear structure) developed by Vary and his collaborators [1,2,3]. In MFDn, the nuclear Hamiltonian is evaluated in a large harmonic oscillator many-body basis and diagonalized by iterative techniques to obtain the low-lying eigenvalues and eigenvectors. The eigenvectors are then used to evaluate a suite of observables which can be compared to experimental quantities. One key feature of the quantum many-body calculations is the rapid increase of basis space dimension with the total number of particles and with the number of harmonic oscillator quanta allowed. The dimension of the basis space characterizes the size of the many-body matrix used to represent a nuclear many-body Hamiltonian. In general, the larger the basis set, the higher the accuracy of the energy estimation and other computable quantities one can obtain [4].

Despite the large dimension of the Hamiltonian matrix produced in nuclear structure calculations, it is sparse, meaning the matrix contains a large number of entries that are zero. The computational method used in MFDn to solve the matrix eigenvalues problem takes advantage of the sparsity structure of the Hamiltonian. The large dimension and the irregular sparsity structure of the Hamiltonian matrix pose a significant challenge to the algorithmic design, data structure specification, parallelization, and memory management strategies in MFDn for large-scale distributed-memory computer systems. Furthermore, the sparsity pattern of the matrix is not known in advance. It is determined efficiently during runtime by MFDn [2].

The original design of MFDn takes into account the standard parallel computing issues such as communication and load balancing [1]. The code has run successfully on as many as 30000 CPU's. The total number of processors on which MFDn is run is restricted to $\frac{n(n+1)}{2}$, where n are odd numbers. These, n processors are referred to as the "diagonal processors". MFDn code has evolved significantly [2,3], but one issue that has not been addressed till now is the I/O performance. In this paper, we describe the benefits of using parallel libraries for performing the I/O. Apart from performance issues, there are other features provided by the parallel libraries that make it more useful for MFDn.

1.1 Overview of MFDn

The MFDn [1] parallel code is used for large-scale nuclear structure calculations in the No-Core Shell Model (NCSM) formalism [5,6], which has been shown to be successful for up to 16-nucleon problems on present day computational resources. It is also used successfully in No-Core Full Configuration (NCFC) applications to a similar range of nuclei [4]. The MFDn code is charged to compute a few lowest (≈ 15) converged solutions, that is, the wave functions and eigenvalues, to the many-nucleon Schrödinger equation:

$$H\,|\phi\rangle = E\,|\phi\rangle \tag{1}$$

The wave functions are expressed as a linear superposition of Slater Determinants in the harmonic oscillator basis. The limit on the retained Slater Determinants is defined by N_{max}, the total number of oscillator quanta above the lowest configuration. These wave functions are then used to calculate a set of observables. The matrix H in (1) is the Hamiltonian operator, which is typically solved using Lanczos diagonalization since H is symmetric and sparse. However, the Lanczos iterative process may be very expensive due to the large dimension of H with many off-diagonal elements. The number of Lanczos iterations also increases significantly to converge the energy levels above the ground state. For example, for the ^{16}O nucleus in the $N_{max} = 6$ basis space, the ground-state energy level requires only 35 Lanczos iterations, while 14 excited states need at least 500 Lanczos iterations to converge. Note that, in this case, the Hamiltonian H has the dimension of 26,483,625.

MFDn constructs the many-body basis space on the n diagonal processors, evaluates the Hamiltonian matrix elements in this basis on $\frac{n(n+1)}{2}$ processors using efficient algorithms [2], diagonalizes the Hamiltonian to obtain the lowest eigenvectors and eigenvalues, then post-processes the wave functions to obtain a suite of observables for comparison with experimental values. The diagonalization produces the wave functions distributed over n diagonal processors. These wave functions are then written to disk and read from disk in subsequent subroutines to verify numerical convergence of the diagonalization procedure and to calculate a suite of observables. Since the diagonal processors hold the wave functions, it is desirable that the I/O of the wave function file is done directly by the n diagonal processors. With heavier nuclei and larger N_{max} value, the size of the wave function file becomes very large and requires a substantial amount of I/O [4]. The dimension of the many-body basis for various nuclei and N_{max} values is tabulated below, along with the total number of processors selected, the total number of diagonal processors, and the wave function file size. Near term plans include matrices in the range of 10-20 billion dimension. The file size follows the growth in dimensions and thus provides scope for introducing parallel libraries to do this I/O. The wave function file is written and read once in a normal MFDn run. This file contains typically 15 eigenvectors of the size as indicated by the *Dimension* column in Table 1.

Table 1. Wave function dimensions and file sizes for a range of realistic test problems

Nucleus	N_{max}	Dimensions	CPU Count($\frac{n(n+1)}{2}$)	Diagonals(n)	File Size(GBytes)
^{12}C	6	32,598,920	190	19	1.82
^{6}He	14	155,710,094	4,950	99	8.70
^{12}C	8	594,496,743	8,646	131	33.22
^{16}O	8	996,878,170	12,090	155	55.70
^{14}F	8	1,990,061,078	27,730	235	111.20

1.2 Motivation for Using Parallel Libraries

I/O has been identified as a major bottleneck in parallel codes and there is a substantial interest in using parallel libraries that can make use of the underlying file system on parallel machines. Two widely used I/O libraries are the HDF5 library [7], and the NetCDF Library [8]. NetCDF is mostly used for array-oriented data whereas HDF5 is also used for storing raster images, complex scientific data and multidimensional arrays. Both libraries have a parallel implementation built on top of MPI-IO, supporting access to files in parallel. In addition to improved I/O efficiency, the use of these parallel I/O libraries has advantages such as portability, human readable files, and self-describing file formats.

In MFDn, the size of the basis space, and thus of the wave functions, becomes very large with heavier nuclei and larger N_{max}, as can be seen from Table 1. Construction of the Hamiltonian matrix and the diagonalization of the matrix using a Lanczos algorithm are the most time-consuming operations in MFDn; however, sequential I/O of the wave functions will become a bottleneck as the dimension of the basis space increases. After diagonalizing the Hamiltonian matrix, the wave functions are available at the diagonal processors. In the current version of MFDn, this I/O takes place sequentially with diagonal processors exchanging the data with the root processor and the root processor performing the actual I/O. We have modified this part of I/O in MFDn to make it parallel by having each diagonal processor write its portion of the data to a single file. The detailed implementation of this will be presented in section 2.2. Since the files are self describing and platform independent, they can be easily archived and used in the future by other groups.

Another advantage of using HDF5 is that data can be added anytime in the HDF5 file. In MFDn, the wave functions previously written into the file are read from the file and the correctness of the data is verified. Furthermore, certain properties like binding energy, total spin and isospin corresponding to each wave function are calculated. It is useful to store these properties along with the wave function, but in a raw binary file, this information cannot be appended to each wave function and had to be added at the end of the file. With HDF5, the space for these properties could be reserved and written later when these properties are calculated. In addition to the wave functions, it is essential to write related data such as a self contained description of the many-body basis space to disk in a portable format. This information can be further used to calculate certain observables outside of MFDn which involves post-processing of the wave functions. Thus, portability and self-describing information are desired properties since this post-processing can be done at a later time on another machine by other researchers. In addition to this, parallel I/O can also be useful for the input of the interaction files to MFDn, in particular the 3-body interaction files which are extremely large.

Node failures, segmentation faults and hardware maintenance interruptions are a stark reality in supercomputing. For large jobs running for many hours, a failure or exceeding wall clock time can result in non-normal job termination. In many cases this results in a major loss of resources. In such cases, it is imperative that large jobs have checkpointing enabled, so that a checkpoint file is created at regular intervals. In cases of abrupt node failures and machine outages, a checkpointed job could be restarted from the last checkpoint file. Portability of the checkpoint file is not very useful for MFDn since it is not feasible to move very large checkpoint files across machines for restarting MFDn jobs. Checkpointing for MFDn is under investigation using MPI-IO.

In MFDn, it may be valuable to save the sparsity structure of the matrix in a self describing format, to facilitate restarts of MFDn with different input data for the same nucleus. The use of parallel I/O libraries for large output files and its potential advantages in checkpointing serve as major motivations for using parallel I/O libraries over serial I/O. This paper focuses on performance improvements using self describing portable formats which we do not contemplate for checkpointing.

1.3 Wave Function File

In this section, the hierarchical structure of the wave function file created with HDF5 is described. To get a better understanding of the file structure, we briefly describe the file structure of an HDF5 file. A file is only a container for storing scientific data. The actual data is in fact stored in other objects inside the file. There are two primary objects in a file that contain the actual data, namely Groups and Datasets. Additional data like attributes and storage properties needed for data organization are also a part of the HDF5 file. Groups are similar to directories and serve as parent directories to datasets, attributes and other sub-groups. Datasets are the actual objects that contain the data of interest. Each dataset set contains the required data array and related meta-data. This meta-data is the data required for self description of files as well as data needed to maintain portability of files. Another very important object of an HDF5 file is dataspace. Dataspace in itself does not contain any data, but is a permanent part of dataset definition. Each dataset has an associated dataspace that contains the rank and dimensionality of the dataset. With this basic understanding of the file structure, we can now understand the wave function file of MFDn. Figure 1 shows the structure of the wave function file in MFDn where the lowest eigenvectors (≈ 15) are stored in different groups with each eigenvector written in parallel by all the diagonal processors.

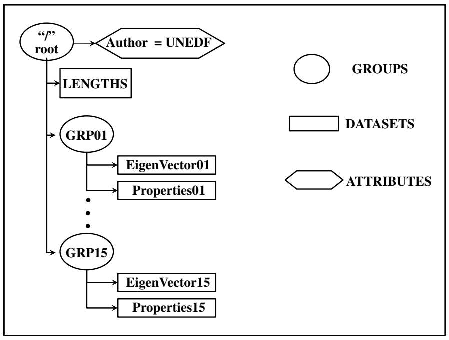


Fig. 1. Wave Function File

The rest of the paper is organized as follows. In section 2 we give an overview of HDF5 for MFDn. In this section, we discuss the differences between two modes of parallel I/O and situations in which these modes are useful. The integration of these parallel libraries with MFDn is also discussed in this section. In section 3, we explain the various experiments and I/O simulations that we have performed using HDF5. This is followed by the performance results in section 4. Finally, we conclude in section 5.

2 Using Parallel HDF5 in MFDn

Both parallel HDF5 and parallel NetCDF libraries provide a high level API for creating self-describing portable files, and both libraries support a wide range of operations. We use HDF5 for the experiments presented here since it appears to be sufficiently flexible and has a large user group. We do not investigate

NetCDF in detail. Parallel HDF5 has both an MPI-IO implementation and a POSIX implementation [9]. There are two modes of I/O that can be performed while doing parallel I/O, namely Independent I/O and Collective I/O.

Some parallel implementations work with the one file per processor approach in parallel I/O. This approach is the fastest but it is prohibitively expensive for programs running on large number of processors, since it requires expensive post-processing. Another method is to do I/O into a single shared file which can be written to and read by all the processors. This is the recommended way of performing parallel I/O and is supported both in NetCDF and HDF5. The parallel libraries provide a single file image to multiple processors, and multiple processors can do parallel I/O in the shared file. Each file is opened with a communicator that is passed as a parameter while creating the parallel file. Once created, a file handle is returned and all future accesses are performed using this handle. Once a file is opened by processes within the communicator, the file is then accessible to all the processors within the communicator. Thereafter, all objects within the file can be modified by these processors.

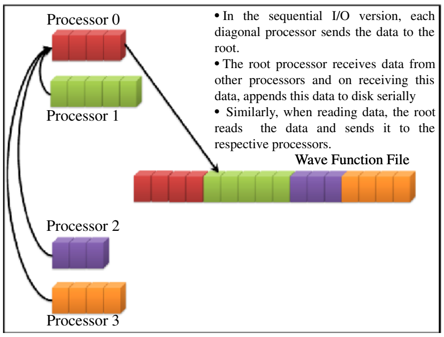
2.1 Performance with Collective and Independent I/O

Parallel I/O can be performed in two ways, collective I/O and independent I/O. Independent I/O means that each process can do the I/O independent of the other processes. Once a file is opened in parallel using a communicator, the file can then be accessed and modified independently by any of the processes in the communicator. In collective I/O however, all processes have to participate in I/O. Since all processes are required to participate, there is obvious inter-process communication required. However, in most cases, multiple I/O requests are interleaved as a single read/write operation, yielding very high speedups. In case of contiguous data layout in which each processor has a contiguous selection, if the selection of each processor is larger than the buffer size of the I/O agent, then independent I/O and collective I/O would both offer approximately the same performance. This occurs because in contiguous data, the data layout is already optimized and collective I/O cannot perform any more optimizations to improve I/O. Moreover, because of the inter-process communication in collective I/O, the performance of independent I/O is better compared to collective I/O since it does not have the overhead of inter-process communication. On the other hand, if each processor has a non-contiguous selection, then the interleaved access requests of different processes can be combined into a single contiguous I/O operation yielding a very high speedup with respect to independent access. That is, with independent I/O each processor has to perform several reads and writes in order to write the whole data leading to high cost of latency [10]. Apart from data layout, I/O performance is also affected by other factors such as caching, network bandwidth, latency, and the file system used.

HDF5 has another driver for parallel I/O besides MPI-IO. This driver is the MPI POSIX driver and is a "combination" of MPI-IO and posix I/O driver [9]. MPI is implemented for coordinating the actions of several processes and posix I/O is used to perform the actual I/O to the disk. This implementation does not support collective I/O mode. In independent I/O mode, MPI POSIX driver may perform better than MPI-IO driver [9], but in our experience the results obtained were identical with both posix and MPI-IO.

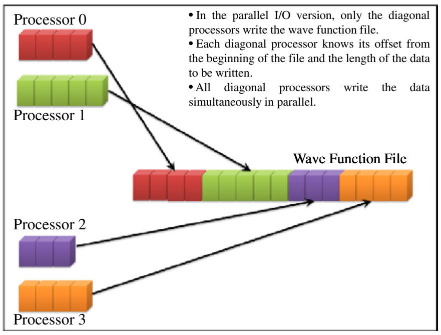
2.2 Integration with MFDn

Currently, the data is written to disk only by the root processor. The data is spread across the n diagonal processors and each diagonal processor sends the data to the root. The root writes its portion of the data and then receives the data from other processors, and then writes it to the disk. This process has obvious communication costs associated with it. Moreover the root processor takes all the load of writing to disk. We parallelized this task with the use of parallel I/O libraries. Since each diagonal processor has a part of the data that needs to be written to disk, that diagonal processor itself writes it to disk eliminating the cost of communication. The offset at which each diagonal processor writes the data is calculated during runtime in MFDn. Figure 2 shows the older sequential write pattern in MFDn and Figure 3 shows how parallel I/O is achieved. HDF5 uses the hyperslab model for doing I/O where hyperslabs are simple subsets of the dataspace. All the diagonal processors define their own non-overlapping hyperslab in the dataset and write the data into the file at specific offsets. To have a very simple and intuitive interface for writing files in HDF5 format, we have created two wrapper functions to read and write the data to file. The interface is straightforward and minimal HDF5 parameters are required while calling the routines for writing and reading the data. Separate modules for HDF5, MPI-IO, and raw binary modes of I/O will be created and, depending on the user choice, an appropriate I/O model will be used.



• In the sequential I/O version, each diagonal processor sends the data to the root.
• The root processor receives data from other processors and on receiving this data, appends this data to disk serially
• Similarly, when reading data, the root reads the data and sends it to the respective processors.

Fig. 2. Sequential I/O



• In the parallel I/O version, only the diagonal processors write the wave function file.
• Each diagonal processor knows its offset from the beginning of the file and the length of the data to be written.
• All diagonal processors write the data simultaneously in parallel.

Fig. 3. Collective I/O

3 HDF5 Performance Tests with MFDn

A normal run of MFDn requires $\frac{n(n+1)}{2}$ processors. After diagonalization of the Hamiltonian matrix, the wave functions are available on the n diagonal processors, and written to disk; the other processors are idle during the actual I/O of the wave functions. In order to test the I/O performance, we have created a suite of test programs to simulate the wave function I/O phase of MFDn on the n diagonal processors. In addition, we have a set of programs that can be used to generate the required input files. Once the raw binary and HDF5 files are available, the I/O performance is measured by a set of programs to simulate posix I/O and HDF5 I/O.

4 Performance Results

In this section, we report the I/O performance using the HDF5 library, as well as the sequential one processor I/O using our I/O test programs. The testbed for our experiments was the super computing Franklin cluster at NERSC. Franklin is a distributed-memory parallel system with 38,640 processor cores available for scientific applications. It has 9,660 compute nodes and each consists of a 2.6 GHz quad-core AMD Opteron processor with a theoretical peak performance of 9.2 GFlop/sec. Each compute node has 8 GBytes of memory. The parallel file system on Franklin is the Lustre file system. Figures 4-6 show the performance of parallel HDF5 for different file sizes. The processors on the X-axis represent the n diagonal processors. For same file sizes, the actual MFDn code would run on

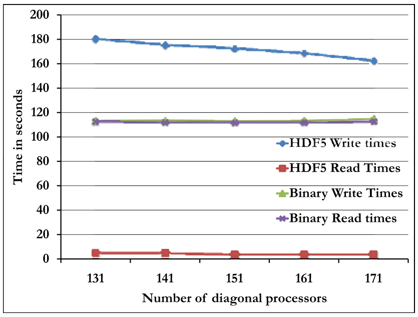


Fig. 4. I/O Performance for 33 GB File

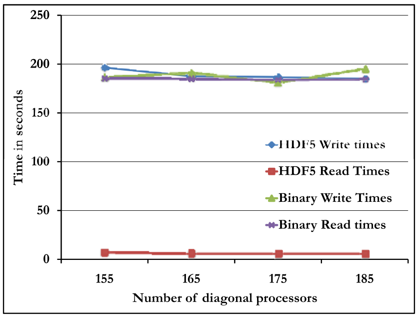


Fig. 5. I/O Performance for 55 GB File

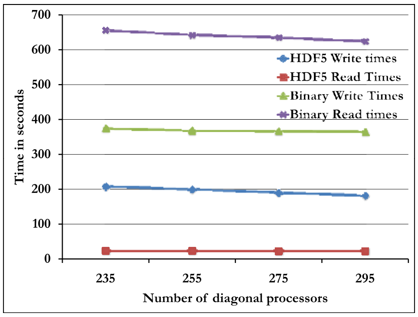


Fig. 6. I/O Performance for 111 GB File

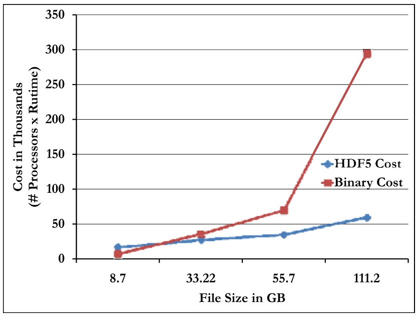


Fig. 7. Cost of using Parallel I/O

$\frac{n(n+1)}{2}$ processors for several hours. It is not feasible to run MFDn code for hours on large scale machines for observing I/O performance. Hence we demonstrate the results using the I/O test programs that we developed.

As can be seen in the plots, the HDF5 write operations are much slower as compared to read operations. The Lustre file system has write locks wherein each writer process acquires a lock on the file and this locking infrastructure effectively prevents simultaneous parallel writes. The parallel write operations are serialized due to the file locking resulting in poor write performance. On parallel file systems such as PVFS, where there are no write locks, the performance of parallel writes can be expected to be much better. Figure 7 shows the cost of using parallel I/O. We define cost as the product of the total number of processors and the total time for I/O. For small files (below 20 GB), parallel HDF5 I/O is more costly than the current sequential binary I/O implementation, but once the file size increases above 20 GB, parallel HDF5 becomes more cost-effective than sequential binary I/O. For the largest file sizes considered here, 111 GB, the difference between parallel HDF5 and sequential binary is approximately a factor of 5 in favor of HDF5. This means that despite locking infrastructure and the slow HDF5 write operations, the cost of using HDF5 is lesser compared to raw binary I/O for large datasets. An output file of about 20 - 100 GB is typical of our large runs where I/O is a significant factor.

We have done all experiments using independent I/O mode since, for contiguous data, independent mode is faster than collective mode. There are several HDF5 users [11], but most of them use HDF5 for complex scientific data in collective I/O mode. According to our knowledge, this is the first real-world application using HDF5 for contiguous data in independent I/O mode.

The I/O performance of any parallel library is also affected by the parallel file system on the clusters. With the Lustre file system, multiple processes cannot write to the same file at the same time. Each time a process has to write, it acquires a lock on the file. So when a new process has to write anything, it sends a request for lock revocation. Only when the lock has been released by the previous writer, can the new process acquire a lock on the file. These are all costly operations and this is the reason for the high write times we can see in the performance charts above. As a work around of this defect in the Lustre and MPI-IO interface, MPI-IO hints can be passed to HDF5 via the MPI_INFO parameter. The *cb_nodes* parameter indicates the total number of writers to be used. By passing a value of "1" to this parameter, we can redirect all the I/O through a single processor. This does not address the root problem but tries to reduce the costly operations of acquiring and revoking locks since all data is written by a single writer. With parallel reads however, there are no locks and this is not the problem, leading to much faster reads. In spite of the slow parallel writes, we intend to have a common I/O approach for all parallel file systems. Computer scientists are currently working to address the problem of MPI-IO and Lustre interface, so we expect better I/O performance from newer version of MPI-IO and Lustre interface.

5 Conclusion and Future Work

We have identified sequential I/O as a bottleneck in the MFDn code performing nuclear structure calculations. We have successfully implemented parallel I/O for the optimization of the MFDn code. The results obtained encourage the use of parallel I/O libraries for sufficiently large datasets. Even for file systems that have a locking infrastructure for parallel write operations, the cost of using parallel I/O is less compared to sequential I/O for sufficiently large datasets. Our contribution is to show how parallel I/O libraries can be of great value to scientific applications dealing with large datasets. We have investigated the difference between collective and independent I/O and situations in which they are effective. With these results in hand, we plan to implement parallel I/O for other large datasets used by MFDn.

Acknowledgments

The work was supported in part by Iowa State University under the contract DE-AC02-07CH11358 with the U.S. Department of Energy, by the U.S. Department of Energy under the grants DE-FC02-07ER41457 (UNEDF SciDAC-2) and DE-FG02-87ER40371 (Division of Nuclear Physics), and by NERSC.

References

1. Vary, J.: The Many-Fermion Dynamics Shell-Model Code. Iowa State University (unpublished) (1992)
2. Sternberg, P., Ng, E.G., Yang, C., Maris, P., Vary, J.P., Sosonkina, M., Le, H.V.: Accelerating configuration interaction calculations for nuclear structure. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Conference on High Performance Networking and Computing, Austin, Texas, November 15 - 21, 2008, pp. 1–12. IEEE Press, Piscataway (2008), http://doi.acm.org/10.1145/1413370.1413386
3. Sosonkina, M., Sharda, A., Negoita, A., Vary, J.P.: Integration of Ab Initio Nuclear Physics Calculations with Optimization Techniques. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008, Part I. LNCS, vol. 5101, pp. 833–842. Springer, Heidelberg (2008)
4. Maris, P., Vary, J.P., Shirokov, A.M.: Ab initio no-core full configuration calculations of light nuclei. Phys. Rev. C 79, 014308 (2009) arXiv:0808.3420 [nucl-th]
5. Navratil, P., Vary, J., Barrett, B.: Properties of ^{12}C in the ab-initio Nuclear Shell Model. Phys. Rev. Lett. 84, 5728 (2000)
6. Navratil, P., Vary, J., Barrett, B.: Large-basis ab-initio No-core Shell Model and its application to ^{12}C. Physical Review C62, 054311 (2000)
7. The HDF Group, http://www.hdfgroup.org
8. The NetCDF Homepage, http://www.unidata.ucar.edu/software/netcdf/
9. Yang, M., Koziol, Q.: Parallel HDF5 Hints. NCSA HDF group
10. Chilan, C.M., Yang, M., Cheng, A., Arber, L.: Parallel I/O Performance Study With HDF5, A Scientific Data Package. In: TeraGrid 2006: Advancing Scientific Discovery, June 12-15 (2006)
11. HDF5 USERS, http://www.hdfgroup.org/HDF5/users5.html