2010

Input/Output of Ab-initio Nuclear Structure Calculations for Improved Performance and Portability

Nikhil Laghave
Iowa State University

Input/Output of ab-initio nuclear structure calculations for improved performance and portability

by

Nikhil Laghave

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Masha Sosonkina, Co-major Professor
James P. Vary, Co-major Professor
Simanta Mitra

Iowa State University

Ames, Iowa

2010

DEDICATION

I would like to dedicate this thesis to my family without whose support I would not have been able to complete this work. I would also like to thank my friends for their help during my research and the writing of this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

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.

I would like to take this opportunity to express my thanks to those who helped me with various aspects of research pertaining to this thesis. First and foremost, Dr. Masha Sosonkina and Dr. James P. Vary for their guidance, motivation and support throughout the research. Weekly research meetings and discussions of the ongoing research work gave me creative insights and deep understanding of the academic material and the significance of the research being involved in. Dr. Masha Sosonkina provided complete support throughout my work and I would like to thank her for giving me this opportunity to work at Ames Lab and for her constant guidance and patience. She has constantly been a source of motivation and it was a privilege working with her. I would like to thank my committee member Dr. Simanta Mitra for his guidance. I would also like to give a special vote of thanks to all the members of the nuclear physics research group. Particularly, Dr. Pieter Maris, Dr. Andrey Shirikov and Alina Negoita for their patience in explaining nuclear physics literature related to my work. I would also like to extend thanks to all my friends for their constant support and motivation.

Last but not the least, I would like to thank my parents for their constant love, support and encouragement.

ABSTRACT

Many modern scientific applications rely on highly computation intensive calculations. However, most applications do not concentrate as much on the role that input/output operations can play for improved performance and portability. Parallelizing input/output operations of large files can significantly improve the performance of parallel applications where sequential I/O is a bottleneck. A proper choice of I/O library also offers a scope for making input/output operations portable across different architectures. Thus, use of parallel I/O libraries for organizing I/O of large data files offers great scope in improving performance and portability of applications. 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. 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 I/O of large datasets along with their portability and ease of use in the down-stream processing. Even situations where the amount of data to be written is not huge, proper use of input/output operations can boost the performance of scientific applications. Application checkpointing offers enormous performance improvement and flexibility by doing a negligible amount of I/O to disk. Checkpointing saves and resumes application state in such a manner that in most cases the application is unaware that there has been an interruption to its execution. This helps in saving large amount of work that has been previously done and continue application execution. This small amount of I/O provides substantial time saving by offering restart/resume capability to applications. The need for checkpointing in optimization code NEWUOA has been identified and checkpoint/restart capability has been implemented in NEWUOA by using simple file I/O.

CHAPTER 1. OVERVIEW

1.1 Introduction

High performance applications require vast amounts of I/O and the wall clock time of such applications incorporate the I/O times as well. There is substantial research and progress made in the field of high performance computing as machines are becoming faster. The same progress, however, has not been achieved in the field of writing data to disk at superfast speeds. In spite of the growth of faster supercomputers, I/O becomes a bottleneck in the performance of applications. Therefore, improving I/O speeds will definitely boost the overall performance of applications where the I/O costs cannot be ignored. In addition to this, the use of external files is of utmost importance in applications where checkpoint-restart capability needs to be implemented. The log files, which contain the application data, may grow in size exponentially as the application size increases. Therefore, efficient I/O of checkpoint file plays a very important role in checkpoint-restart applications. In this thesis we address the importance of external files for improving the performance of applications doing ab-initio nuclear structure calculations. We illustrate methods employed to improve the performance of MFDn; a state-of-the-art nuclear physics code.

1.2 Overview of Ab Initio Nuclear Structure Calculations

A calculation is said to be ab-initio (or "from first principles") if it relies on basic and established laws of nature without additional assumptions or special models. In nuclear physics, the ab-initio problem is referred to solving for the nuclear properties with use of the best available nucleon-nucleon (NN) potentials, supplemented by 3-body (NNN) potentials as needed, using a quantum many-particle framework that respects all known symmetries of the potentials. The

ab-initio problem is known to be computationally hard and there are three known approaches namely Greens Function Monte Carlo [1], Coupled Cluster [2] and No Core Configuration Interaction (CI) methods [3, 4, 5, 6, 7], that have proven applicable for nuclei with more than 6 nucleons. All three methods are computationally very expensive but all are known to be potentially exact in finding the solution. In this thesis, we discuss the nuclear physics code MFDn [8], for ab-initio CI calculations.

The MFDn parallel code is used for large-scale nuclear structure calculations in the No-Core Shell Model (NCSM) formalism [4, 9], which has been shown to be successful for up to 16-nucleon problems on present day computational resources. MFDn constructs the many-body basis states and the Hamiltonian matrix, and solves for the lowest eigenstates using the Lanczos algorithm. The found eigen-vectors, called wave-functions, are not experimentally observable, so the wave functions are used to evaluate a suite of physical observables which can then be compared to the experimental data.

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 [6].

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 eigenvalue 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 [10].

1.3 Motivation

The MFDn code has been in development for more than 2 decades and is constantly being optimized for better performance. Efficient use of memory was the main focus during the early development of MFDn. Significant improvements have been made over the last three years to improve the performance of MFDn [10, 11, 12]. Over the past three years, the overall performance has improved by a factor of 3 to 4, depending on the processing element (PE) count. MFDn has good scaling properties and runs on thousands of PE's of existing supercomputing architectures. MFDn code has evolved significantly and has run successfully on as many as 30,000 PE's. The original design of MFDn takes into account the standard parallel computing issues such as communication and load balancing [8]. As it continues to evolve, the I/O performance becomes a bottleneck for MFDn and offers a scope for improving it's scalability and performance.

For the work presented in this thesis, the basis set used by MFDn is the Woods-Saxon basis. The Woods-Saxon Potential is a model for the mean field potential for nucleons inside the nucleus. The motivation for trying the Woods-Saxon basis comes from the fact that certain observables sensitive to the long range part of the wave-functions seem slower to converge in harmonic oscillator basis which is the other choice of basis for MFDn. The convergence is slow in the harmonic oscillator basis because the gaussian tails of the harmonic oscillator wave-functions poorly represent the asymptotic exponential tails.

Program robustness requires error handling that anticipates, detects, and resolves errors at run time. The highest level of error handling capability is fault tolerance that attempts to recover application state from hardware or operating system failures if possible, and if not, terminates the program gracefully. Checkpointing is a technique for inserting fault tolerance into computing systems. It basically consists of storing a snapshot of the current application state, and later on, use it for restarting the execution in case of failure. The MFDn code needs to be run several times until suitable parameters are found for optimal W. S. basis. This process of finding appropriate parameters can take several hours depending on the nucleus and computing platform. Batch processing at super-computers and the upper bounds on

these scripts require that there should be some kind of restart capability in the optimization technique employed to do the search of appropriate parameters. This is so because the hard work of finding parameters and running MFDn several times is lost if for some reason, the program is stopped before finding the minimum function value. For heavier nuclei, for example, restart capability becomes inevitable. Therefore, implementing checkpoint-restart capability in a long-running application is one of the foremost challenges from the computer science point of view. This thesis addresses the two issues discussed above: (1) portability of MFDn output files and (2) use of files to save history of applications to allow checkpoint-restarts in MFDn.

1.4 Approach to the Solution

In this thesis, we will present the work addressing the following two challenges in the

1. Portability and parallelization of wave function files.

2. Checkpoint-restart via file I/O with the Newuoa optimization algorithm as example.

The use of parallel I/O libraries can prove to be very important for writing wave function files for portability and human readability. I/O libraries are useful for making the I/O to disk parallelized using high-level library primitives. Parallel I/O would avoid the overhead of communication among diagonal processors and the root processor, and make the I/O to disk more load balanced since all the diagonal processors would be doing the I/O simultaneously. In the MFDn code, where the Newuoa search code is required to find the appropriate Woods-Saxon parameters for finding the minimum ground state energy, a restart capability can be much useful. Having the Newuoa code save all the history to disk and then use this history in case of restarts would significantly lower the runtime, since the function evaluations previously completed by MFDn can be just read from file. Thus checkpointing is a viable solution to the problem of running very long searches which would span a period of several days. We have implemented checkpointing in the Newuoa code by using I/O of checkpoint files and parallelized the I/O of the wave functions by using parallel I/O libraries.

1.5 Thesis Layout

The rest of the thesis is organized as follows:

Chapter 2

In Chapter 2, we discuss the relevant literature related to this thesis. A brief introduction to the importance of parallel I/O libraries and details of some popular I/O parallel libraries are given. This is followed by an introduction to checkpointing and its various types. The chapter is concluded by providing a brief introduction on optimization techniques.

Chapter 3

This chapter details structure of the nuclear physics application suite doing no-core interactions. A brief description about the relevant algorithmic implementations of MFDn are given. This chapter also explains the role of various upstream codes and a description of the various phases of MFDn execution which are together responsible for doing the ab-initio nuclear physics calculations. An introduction to Newuoa; which is the choice of optimization technique for the work done in this thesis, is also given. This chapter explains how different programs fit into the workflow to perform ab-initio calculations.

Chapter 4

Chapter 4 begins with a description of how parallel I/O libraries are important for high performance computing. An abstraction of how different components fit into the I/O stack is presented and the operation of the each layer in this abstraction is explained. This chapter also discusses MPI-IO in brief due to the fact that of all the layers of the I/O stack, the MPI-IO layer is the foundation on which parallel I/O libraries are built. This is followed by a a brief description of the netCDF and HDF5 I/O libraries.

Chapter 5

Finally in chapter 5, we discuss the implementation of parallel I/O using the HDF5 library. This includes the in-depth explanation of the two parallel implementations currently part of the HDF5 library, namely, Collective I/O and Independent I/O. We discuss the reasons for our choice of implementation. This is followed by the implementation part of parallel I/O in MFDn. The difference in serial and parallel implementations of wave-function I/O is explained. The chapter is concluded by presenting the performance results obtained by using parallel I/O. The cost comparison of binary and HDF5 I/O is demonstrated, which clearly shows that after a certain threshold, HDF5 outperforms raw binary I/O.

Chapter 6

In this chapter, the complete application suite running upstream codes, MFDn and downstream codes is discussed. The role of optimization algorithm: NEWUOA is explained as a part of this workflow. The need for application checkpointing in NEWUOA is explained followed by showing how checkpointing is implemented in NEWUOA. The interface changes due to checkpointing implementation are explained.

Chapter 7

Chapter 7 summarizes our research, concludes the thesis and mentions the future direction that this research can take.

CHAPTER 2. Literature Review

2.1 Parallel I/O Libraries

High Performance Computing systems are usually compared in terms of their floating point performance, speed of communication and memory. However as applications become larger and larger, they have to deal with large amounts of data and this data slowly becomes as much as a bottleneck as the actual computation. As I/O has been identified as a major bottleneck in parallel codes, there is a substantial interest in using parallel libraries that can make use of the underlying file system on parallel machines. Ideally the interfaces provided by the I/O libraries should be both portable and standardized if it has to achieve widespread acceptance.

The most significant effort in this area is the MPI-IO [13] standard that has now been incorporated into the latest version of MPI i.e. MPI-2 [14]. Two widely used I/O libraries that are built on top of MPI-IO are the HDF5 library [15], and the NetCDF Library [16]. NetCDF is mostly used for array-oriented data whereas HDF5 is also used for storing raster images, complex scientific data and multi-dimensional 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 have 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 3.1. 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. This I/O of wave-functions can be made parallel by using I/O libraries which also have other advantages such as portability and human readability of files.

Parallel libraries give the capability of adding data at any time in the output files. 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 have to be added at the end of the file. Space of these properties can be allocated within the file using parallel I/O libraries. These properties could thus be written later as and when they 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. It may also 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.

2.2 Checkpointing

Checkpointing is a technique of inserting fault tolerance into computing systems. It basically consists of storing a snapshot of the current application state, and later on, use it for restarting the execution in case of failure. Node failures, segmentation faults and hardware

maintenance interruptions are a stark reality in supercomputing. For large jobs running for many hours, a hardware 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 check-pointed 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.

An area in which checkpointing has a potential for becoming very important for MFDn is the search of optimal Woods-Saxon parameters for finding the least ground state energy. This is implemented using suitable derivative free optimization techniques. These optimization techniques evaluate some given objective function $F(x)$, where $x \in R^n$, using initial values of input parameters. The optimization technique will continue doing iterations (MFDn evaluations) until it has found the local minimum of the objective function or some terminating criteria is met. For use with MFDn, the input parameters given to the optimization technique are the Woods-Saxon parameters and the objective function value is the ground state energy obtained at the end of MFDn execution. For heavier nuclei and larger N_{max}, the search of parameters will become very time consuming and will span several days. Currently the searches are shorter and can be completed within the times allowed for batch scripts on supercomputers. However in future, batch jobs would run for days and it is not always feasible to run the same job for several days. Hence it is imperative that the optimization technique for searching parameters has checkpointing enabled so that long jobs can easily span many days. This feature is also favorable for jobs that are usually run in one batch. One batch jobs may run fast and produce the optimal parameters faster but there is always a possibility of these jobs getting interrupted due to node failure, hardware maintenance, segmentation faults etc. In that case checkpointing will allow the jobs to resume the execution from the last logged point instead of the beginning. Thus checkpointing has multifold advantages, namely resuming jobs interrupted due to some hardware maintenance, node failures, segmentation faults as well as resuming jobs which

require multiple batch runs because of the upper bound on time limits on supercomputers. Implementation of checkpointing in one such optimization technique, NEWUOA is presented in this thesis.

There are three levels under which checkpointing method can be categorized [17]. They differ in level according to the involvement of the programmer.

1. **OS Checkpointing:** Here, checkpointing is implemented by the operating system. In case of OS checkpointing, any program can be checkpointed by the operating system without any involvement from the programmer. A simple example of OS checkpointing is the standard process pre-emption i.e. making a process relinquish the CPU and putting itself on the ready queue after saving its current state. Most operating systems do not implement checkpointing beyond process scheduling.

2. **User-Level, Transparent Checkpointing:** In this case, the checkpointing is performed by the program without help from the operating system. Transparency is achieved by either compiling the application program with a special checkpointing library or by rewriting the executable files.

3. **User-Level, Non-Transparent Checkpointing:** In this case, programmers incorporate checkpointing into their programs, often by using checkpointing libraries and preprocessors. This process places a much larger burden on the programmer. Non-transparent checkpointing has its advantages in terms of performance and flexibility. Programmers usually specify the exact amount of information needed for recovery and therefore save less information than the transparent checkpointing method. Moreover, with non-transparent checkpointing, programmers can save the checkpoint files in portable formats which allows the checkpoints to be restored on machines with different architectures. This is not possible in case of transparent checkpointing.

The work in this thesis adopts an User-Level Non-Transparent checkpointing method which saves and recovers the function evaluation history using file I/O.

2.3 Optimization Techniques

In this section we discuss some optimization techniques which are prevalent in the areas of computing. Many scientific and engineering arenas require a good combination of parameters to optimize some performance metric or cost function. Modern nuclear physics practices require use of large, sophisticated, computer intensive calculations of cost function (CF) to accurately model the interactions of a nucleus. Frequently, these CFs are not analytic, but are obtained from simulation, experiments, or from a series of numerical computations [18]. Generally the CF is not smooth and multiple local optima are often present. Derivatives are usually not available in closed form, and are difficult to calculate numerically. New optimization techniques for complex cost functions have recently become an active research area, especially for global and large-region searches [19]. With the increasing availability of parallel computing systems, high-performance approaches are now employed to address the high computational cost of these new optimization techniques [20, 21, 22]. An optimization problem can be represented as:

Given: A function $f : A \to \mathbb{R}$ from the set A to the real numbers $\mathbb{R}$.

Find: An element x_0 in A such that $f(x_0) \leq f(x)$ for all x in A ("minimization") or such that $f(x_0) \geq f(x)$ for all x in A ("maximization").

Such a formulation is called an optimization technique. Optimization techniques can be broadly classified under two categories: Single Variable Optimization(SVO) and Multiple Variable Optimization(MVO). Thereafter, optimization algorithms can be sub-classified as Linear Programming, Integer Programming, Heuristic Algorithms, Combinatorial Optimization, Dynamic Programming and many more. Some of the problems formulated using this technique in the domain of computer science and physics may refer to the technique as energy minimization where energy represents the value of the function f of the system being modeled. Typically, A is some subset of the Euclidean space $\mathbb{R}^n$, often specified by a set of constraints, equalities or inequalities that the members of A have to satisfy. The domain A of f is called the search space, while the elements of A are called candidate solutions or feasible solutions. The function f is objective function, or CF. A feasible solution that minimizes (or maximizes, if that is the

goal) the objective function is called an optimal solution. Generally, when the feasible region or the objective function of the problem does not present convexity, there may be several local minima and maxima, where a local minimum x^* is defined as a point for which there exists some $\delta > 0$, that for all x the expressions;

$$||x - x^*|| \leq \delta \tag{2.1}$$

$$f(x^*) \leq f(x) \tag{2.2}$$

holds; that is to say, on some region around x^* all of the function values are greater than or equal to the value at that point. Similarly local maxima can be defined as a point in some region around x^* where all the function values are less then or equal to the value at that point. A number of algorithms proposed for solving non-convex problems including the majority of commercially available solvers are not potentially capable of making a distinction between local optimal solutions and rigorous optimal solutions, and will treat the former as actual solution to the target problem. The branch of applied mathematics and numerical analysis that is concerned with the development of deterministic algorithms, capable of guaranteeing convergence in finite time to the actual optimal solution of a non-convex problem is called global optimization.

For twice-differentiable smooth functions, unconstrained problems can be solved by finding the stationary points where the gradient of the objective function is zero and using the Hessian matrix to classify the type of each point. If the Hessian is positive definite, the point is a local minimum, if negative definite, a local maximum, and if indefinite it is a saddle point. However, the existence of derivative is known a priori and there are methods devised for these specific situations. The basic classification of optimization techniques based on the optimization method [23] are:

1. Gradient Based Methods

   For modern non linear functions, Gradient Based Methods are primary optimization techniques. Steepest Descent Method is a classic example of gradient based minimization

technique. The decision of choosing the next point is made by calculating the gradient, $\hat{g}$, of the function at each iteration. Refer to equation 2.3

$$x_{k+1} = x_k + \alpha g_k \tag{2.3}$$

The line search step alpha is added as a refinement to this very simple optimization routine. Alpha is a step size, for the sake of limiting the search step, to prevent large overshoot, or oscillations while searching for the optimum. This type of gradient-based method is a first order method, because we are using solely gradient information. Another most popular method, Newtons Method, introduces second order information in the form of Hessian, equation 2.4.

$$x_{k+1} = x_k + \alpha H_{k^{-1}} g_k \tag{2.4}$$

Other common derivate oriented algorithm is Sequential Quadratic Programming. Gradient based approaches are efficient and are best known methods for local optimization. The problems which this kind of methods can face are as:

- Derivatives are not always available.
- Finite difference approximations are too expensive or inaccurate.
- Objective functions with various local minima or have added noise to it.
- High dimensional problems preclude accurate estimation of gradient [24].

2. Derivative Free Methods

Derivatives Free Methods were amongst the initial optimization methods. They rely on ability to compute function values and make decision based on relationship amongst the values rather than actual numeric values. Two such applications that use derivate free methods are NEWUOA and DIRECT [25]. The advantage of these techniques is that

they do not use gradients to find search directions, so in principle they can deal with noisy problems. Through the years, more and more sophisticated logic has been developed to allow these types of algorithms to intelligently search through the design space. These may be as simple as distributing the search, such as in Parallel Direct Search [26], or using a simple method, as in Boxs Complex Method [27].

The work presented in this thesis is based on integration of a Derivative Free Optimization technique, NEW Optimization Algorithm(NEWUOA) and the ab-inito nuclear physics code MFDn.

CHAPTER 3. No-core Configuration Interaction

3.1 Background

Many Fermion Dynamics nuclear (MFDn) parallel code is used for large-scale nuclear structure calculations in the NCSM formalism. The MFDn code is charged to compute a few lowest (≈ 15) converged solutions, called wave-functions, to the many-nucleon Schrödinger equation: MFDn code requires a set of input files for performing ab-initio calculation. These input files are the Hamiltonian files, the Interaction files and files that provide parameter sets and variable values, such as number of Lanczos iteration etc., for calculation to MFDn code. The output of the MFDn code is a set of text files that have information about theoretical observables, such as excitation energies at various level, their relative nuclear spins and wave-function values.

$$H\,|\phi\rangle = E\,|\phi\rangle \tag{3.1}$$

This calculated excitation energy is matched against experimental energy based on the nuclear spin of each level. The number of processors required for MFDn execution typically depends on the size of the Hamiltonian matrix of the nucleus taken in consideration and hardware platform used for experiment. As the size of the Hamiltonian matrix increases for heavier nuclei and larger N_{max}, the run time of MFDn increases exponentially and MFDn code can run up to hours because of the expensive Lanczos iterations.

Other properties, called observables, are formed from the calculated wave-functions. The matrix H in equation 3.1 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 huge dimensionality of H with many off-diagonal elements. The number of Lanczos iterations also increases significantly for the energy levels beyond the ground

state. For example, for the ^{16}O nucleus in the $6\hbar\omega$ basis space, the ground-state energy level requires only 35 Lanczos iterations, while 15 excited states need at least 300 Lanczos iterations for convergence. Note that, in this case, the constructed Hamiltonian H has the dimension of $26,483,625$. MFDn constructs the $m-scheme$ basis space, evaluates the Hamiltonian matrix elements in this basis using efficient algorithms, diagonalizes the Hamiltonian to obtain the lowest eigenvectors and eigenvalues, then post-processes the wave-functions to obtain a suite of observables and compares them with experimental values.

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 wave-functions are not experimentally observable, so the wave-functions are used to evaluate a suite of physical observables which can then be compared to the experimental data.

MFDn constructs the many-body basis space on the n diagonal processors, evaluates the Hamiltonian matrix elements in this basis on $n(n+1)$ processors using efficient algorithms [10], 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 [6]. 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 3.1.

Table 3.1: 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

In addition to MFDn, there are some other programs called the upstream codes which run before MFDn and prepare some input files for MFDn. The output from upstream codes is then collected into a file named `mfdn.dat` that is the input for MFDn. Then MFDn runs and gives the ground state energy of a certain nucleus under consideration. This ground state energy is further minimized by Newuoa to match with the experimental values. The set of Woods-Saxon parameters that are required by MFDn are determined from Newuoa searches for optimal basis states, where optimal is defined by minimizing the ground state energy of the interacting many-body system.

Once the Woods-Saxon parameters are determined by Newuoa, they are given by a program called `create_dat_files` to an upstream code called `M2` that will determine the Woods-Saxon Potential and therefore, the Woods-Saxon basis. The Woods-Saxon Potential is a model for the mean field potential for nucleons inside the nucleus. The motivation for trying the Woods-Saxon basis comes from the fact that certain observables sensitive to the long range part of the wave-functions seem slower to converge in harmonic oscillator basis. Those observables are weakly bound states, RMS radii, B(EL), B(ML), etc. The convergence is slow in the harmonic oscillator basis because the gaussian tails of the harmonic oscillator wave-functions poorly represents the asymptotic exponential tails.

3.2 Implementation of Ab-Initio Calculations

3.2.1 Upstream Codes

Upstream codes are a set of codes created by nuclear physicists. These codes usually run before MFDn and prepare the necessary input files for MFDn. A brief explanation of the upstream codes is given below.

1. **Heff:** Heff is a F90 code to input 2-body RCM matrix elements of H, or components thereof, in the full space (truncated) and generate Heff (or Veff) in chosen subspace, the P-space, using the method of Wilson-Lee-Suzuki-Okamoto-Okubo [4, 9]. Here, H is the hamiltonian matrix as described in Chapter 2. This code works through direct matrix diagonalization, inversion, and multiplication.

   There are two input files to `Heff`:

   - `Heff_LS.dat` : Parameters of the effective operator such as (fermion number, $h\omega$, P-space)
   - `Bare_inputRCM_matrices` : Bare operator matrix elements.

   And one output file:

   - `Veff_outputRCM_matrices` : Output Matrix.

2. **M2:** M2 are the m2v9 codes that transform some operators such as $T_{rel}, V_{coul}, H_{rel}, H_{cm}$ from Harmonic Oscillator (HO) basis into Woods-Saxon (WS) basis.

3. **GOSC:** The format of the files created by the previous codes is not the same as that is required by MFDn. The files need to be reformatted according to MFDn. GOSC is used to prepare a certain format for those input files as required by the MFDn code. The output from these upstream codes that run before MFDn is then collected into a file called `TBME_Veff.int` that is the input for MFDn.

3.2.2 Many-Fermion Dynamics Code

MFDn parallel code is used for large-scale nuclear structure calculations in the No Core Shell Model (NCSM) formalism. We will briefly outline the structure of the MFDn code.

MFDn constructs the many-body basis states and the Hamiltonian matrix H, and solves for the lowest eigenstates using the Lanczos algorithm. It writes the nuclear wave-functions (eigen values) to file, and evaluates selected physical observables such as the ground state energy which can be compared to experimental data. The matrix is distributed as a 2-D array over the processors, as indicated in Figure 3.1 borrowed from [10], and MFDn works with the lower triangle part of this matrix, because the matrix is symmetric. The Lanczos vectors, needed for re-orthogonalization after every matrix-vector multiplication, are distributed over all processors. MFDn runs on $n(n+1)/2$ processors because of the 2-D distribution of the matrix, where n is the number of *diagonal* processors (see Figure 3.1). There are several phases in the code, which we try to explain below.

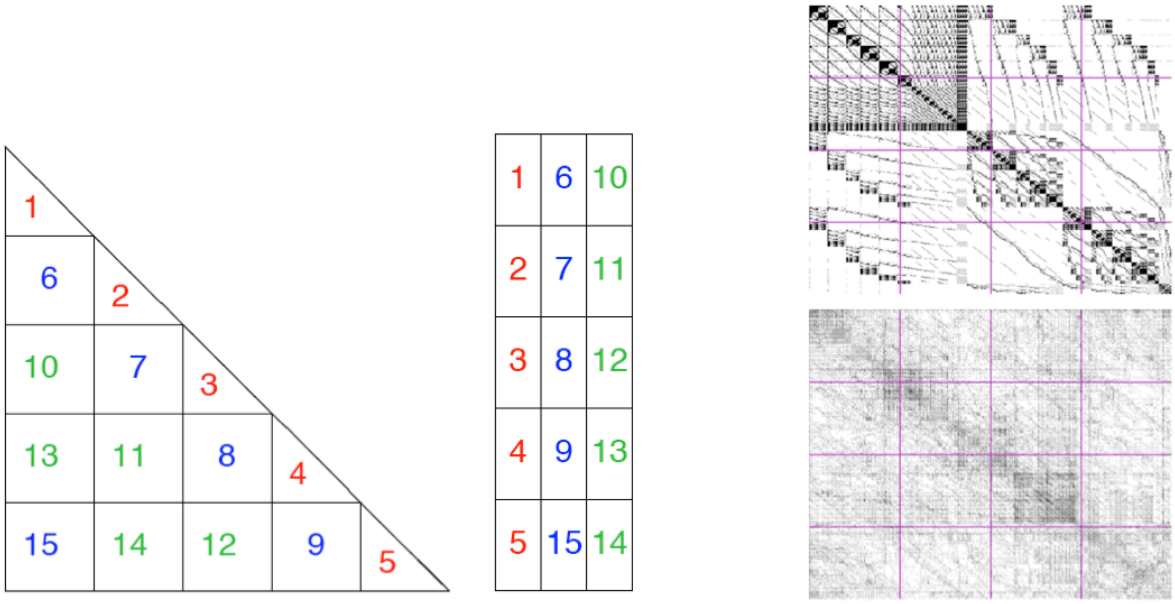

Figure 3.1: Left: Two-dimensional distribution of the lower triangle of the matrix and of the Lanczos vectors over $n(n+1)/2$ processors with $n = 5$. The first n processors (in red) are referred to as the *diagonal processors*. Right: load-balanced distribution of nonzero matrix elements on $n(n+1)/2$ processors with $n = 4$.

1. **Setup of Basis:** The first phase of MFDn is the setup phase where the single-particle basis states are defined, and the A-body basis is developed. The A-body basis is constructed only on the n diagonal processors.

2. **Construction of Matrix:** The next phase is the construction of the sparse, real and symmetric matrix H. A matrix element H_{ij} can only be nonzero if the two A-body basis states i and j differ by at most K single-particle states. With a lexicographical ordering of the A-body basis states on a single processor, neighboring basis states often differ by only one single-particle state, and nonzero matrix elements tend to occur in blocks, see Figure 3.1. A naive 2-D distribution of this matrix leads to poor load-balancing, both in terms of memory needs and in terms of CPU time. With the round-robin distribution of the A-body basis states over the n diagonal processors, neighboring basis states tend to have very few single-particle states in common and the nonzero matrix elements appear to be randomly distributed (right panel of Figure 3.1). However, the round-robin distribution of the basis states makes the construction of the matrix much more cumbersome and time-consuming as the matrix size increases. A *multilevel blocking scheme*, based on groups of single-particles states [10] was introduced to cope with the problem of determining which matrix elements can be nonzero. The performance of the code depends on the particular choice of how to create these groups – the most efficient choice depends not only on the nucleus, but also on the number of processors.

3. **Lanczos Iterations:** Once the matrix H is constructed, stored in memory, using compressed sparse column(CSC) format and distributed over all processors, MFDn solves for the lowest eigenvalues using an iterative Lanczos algorithm. Each iteration of the Lanczos algorithm consists of a matrix-vector multiplication, followed by an orthogonalization against all previous Lanczos vectors. All these previous Lanczos vectors are also stored in memory, distributed over all processors. This phase incurs a lot of MPI communication because of the 2-dimensional distribution of the matrix over all the processors: After each matrix-vector multiplication, the resulting output vector needs to be collected on the n diagonal processors. Next, this vector needs to be distributed to all processors in order to

do the orthogonalization, and finally the new input vector needs to be distributed to all processors. The number of Lanczos iterations are provided using an input variable. After these fixed number of Lanczos iterations, the lowest eigenvalues and the corresponding wave-functions are written to disk from the n diagonal processors. This part provides a scope for parallelizing the I/O of wave-functions which can be implemented using parallel I/O libraries such as MPI-IO, HDF5 and NetCDF.

4. **Evaluation of physical observables:** In the final phase of MFDn, the wave-functions are read back in another subroutine. The wave-functions are used to evaluate physical observables such as the rms radius of the nucleus, its dipole and quadrupole moments, and radiative transitions between the ground state and excited states.

3.2.3 Optimization for MFDn

The ground state energy from the output of MFDn needs to be minimized to match with the theoretical values. Several parameters for the upstream codes need to be searched in order to minimize the objective function value specified by the ground state energy. The set of Woods-Saxon parameters are determined from searches for optimal basis states, where optimal is defined by minimizing the ground state energy of the interacting many-body system.

Upstream codes are a set of codes that usually run before MFDn and prepare the input files required for MFDn. Once these files are created, MFDn runs using input files and interaction files. The ground state energy estimation of MFDn is then used by an optimization search program which attempts to minimize a function value $F(x), x \in \mathbb{R}^n$. VTDIRECT and NEWUOA are examples of such programs which work well and have been tested with MFDn. These optimization techniques run using some initial parameters and try to find the appropriate parameters to minimize the objective function. The new parameters are then used by upstream codes and MFDn to calculate new ground state energy. The output of MFDn is used to calculate the objective function that needs to be minimized. Figure 3.2 shows the flowchart of the entire process.

Once the Woods-Saxon parameters are determined by using an optimization technique,

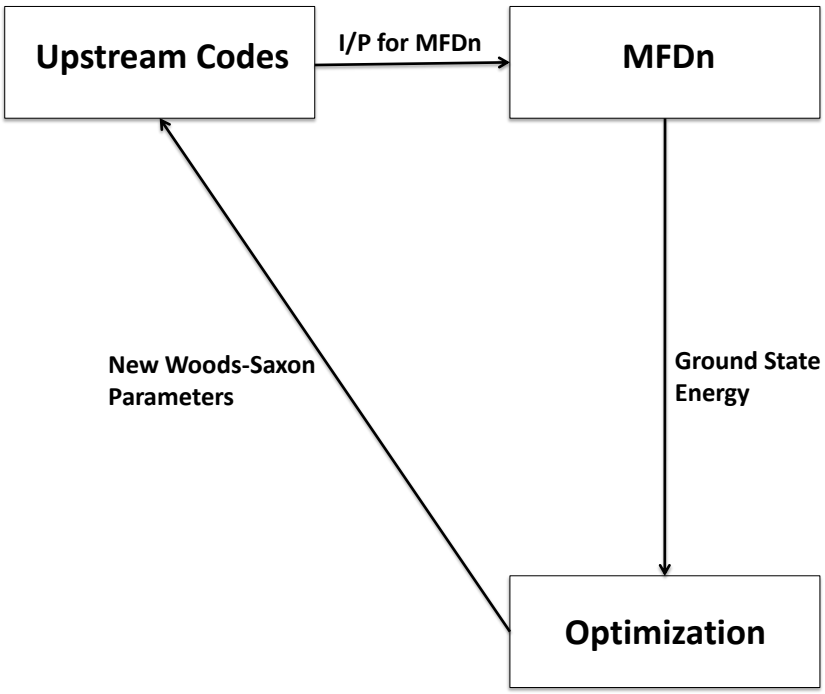

Figure 3.2: Nuclear structure calculations with optimization component

they are given by `create_dat_files` to M2 (m2v9) code that will determine the Woods-Saxon Potential and therefore, the Woods-Saxon basis. The process of matching theoretical and experimental values in order to find a good match typically requires large number of calculations and are thus tedious and error prone. The number of calculations increases to a much higher number when more number of parameters to be searched are introduced in the process. Each iteration of calculation of energy function requires generation of values for different parameters to be searched which can be error prone and is more complex. The NEWUOA [28] software uses quadratic approximations to the objective function that are highly useful for obtaining a fast rate of convergence in iterative algorithms for unconstrained optimization, because usually some attention has to be given to the curvature of the objective function. The quadratic model for such approximations has $\frac{1}{2}(n+1)(n+2)$ parameters and this number of calculations of values of the objective function is prohibitively expensive in many applications with large n. The Newuoa algorithm tries to construct suitable quadratic models from fewer data. The model $\mathcal{Q}(x), x \in \mathbb{R}^n$, at the beginning of a typical iteration, has to satisfy only m interpolation conditions

$$\mathcal{Q}(x_i) = \mathcal{F}(x_i), \qquad i = 1, 2, ..., m \tag{3.2}$$

where $\mathcal{F}(x), x \in \mathbb{R}^n$, is the objective function, the number m is prescribed by the user, and where the positions of the different points $x_i, i = 1, 2, ..., m$ are generated automatically. Newuoa requires $m \geq n+2$, in order that the equation 3.2 always provide some conditions on the second derivative matrix $\nabla^2\mathcal{Q}$, and $m \leq \frac{1}{2}(n+1)(n+2)$, because otherwise no quadratic model $\mathcal{Q}$ can satisfy all the equations 3.2 for general right hand sides.

CHAPTER 4. Parallel I/O libraries

Before going into more detail about parallel I/O libraries, it would be helpful to see how these I/O libraries fit into the bigger picture of scientific high performance computing and application I/O.

4.1 Background

The current leadership computing platforms contain hundreds of thousands of processors. For applications to efficiently utilize these computers, applications make use of simplifying abstractions provided by several tools and libraries. MPI libraries provide a standard programming interface for parallel computation on a wide array of hardware. Similarly, mathematical libraries like BLAS hide the details of CPU-specific optimizations. These lower-level mathematical and communication libraries contribute to an overall software stack built to allow developers to focus on the science behind their applications. The same story follows for I/O performance on high-end computing platforms. I/O performance comes from aggregating many storage devices. In fact, CPU performance has consistently improved at a rate much faster than that of storage devices. Many modern scientific applications rely on highly parallel calculations, which scale to 10's of thousands processors. The same does not apply for storage. This discrepancy makes the need for parallel I/O across multiple devices even more acute. Yet as the number of devices involved in I/O grows, coordinating I/O across those devices becomes more of a challenge. As communication and mathematical libraries assist the parallel computation side of scientific simulations, an "I/O software stack" [29] as shown in Figure 4.1 assists data management for high performance applications on such platforms.

Storage devices form the foundation of the software stack. Several additional layers provide

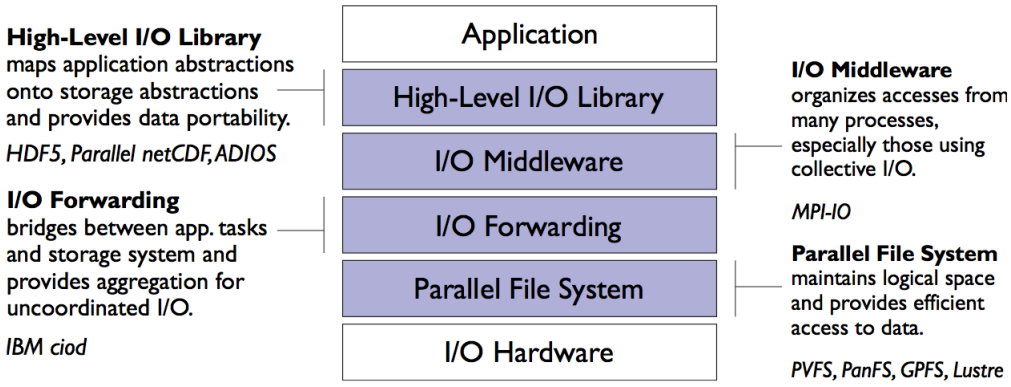

Figure 4.1: I/O Software Stack

abstractions to help make the task of extracting maximum performance accessible to applications programmers. Several I/O libraries available to the scientific programmer hide many lower level details and tend to make the I/O performance efficient. Though hidden by high level I/O libraries, we briefly discuss the lower levels of the software stack which will provide a better appreciation of the usefulness of high level I/O libraries.

- Storage systems contain thousands of individual devices that increase the potential bandwidth. The *parallel file system* manages all the storage devices and represents them into a single logical unit. At this layer, the file system lacks some important features: typically file systems do not have a mechanism for co-ordinated I/O, and the data model is fairly low level for computational scientists to use directly.

- The *I/O forwarding layer* typically exists only on the largest computer systems. Applications do not interact with this layer directly. Rather, this layer simplifies the scalability challenge: a large number of compute nodes communicate with a smaller number of I/O forwarding nodes, and these nodes in turn talk to the file system.

- The *Middleware*, or *MPI-IO*, layer introduces much more sophisticated algorithms specifically tailored for parallel computing. This layer coordinates I/O among a group of processes, introducing collective I/O to the stack. MPI datatypes allow applications to

describe arbitrary I/O access patterns. Applications can use the MPI-IO layer directly, but the programming model is still rather low-level and not a perfect fit to the needs of common applications.

- The *High-Level I/O Library* introduces concepts such as multidimensional arrays of typed data which are better suited to scientific applications. For example, a climate code can represent temperature in the atmosphere with a four-dimensional array: latitude, longitude, altitude, and number of degrees Celsius. Further, these libraries define the layout of data on disk. These self-describing file formats make exchanging data with colleagues and other research groups at present and in the future much easier.

To incorporate parallel I/O into applications, the participation of the programmer only begins from the layer shown as the Middleware Layer in Figure 4.1. As mentioned above, this layer introduces sophisticated algorithms specifically tailored for parallel computing. This layer also coordinates I/O among a group of processes. MPI datatypes make it simpler to describe arbitrary I/O access patterns. Using the MPI-IO layer for the application programmer is quite usable and efficient in spite of the fact that this model is still rather low-level and not a perfect fit to the needs of common applications. We will still discuss the MPI-IO layer in brief as it forms the foundation of the other, simpler to use, flexible and powerful parallel I/O libraries

4.2 MPI-IO

As most scientific applications are more involved in the scientific part of the application, most of them do not concentrate on parallelizing input/output operations. In particular, sequential I/O has been identified as a bottleneck. To make the I/O to disk parallel, I/O libraries supporting parallel I/O can be used. There are several I/O libraries available for this purpose, however HDF5 and netCDF are the most popular and widely used I/O libraries. Both HDF5 and netCDF provide a set of software libraries and data formats that support the creation, access and sharing of scientific data. Because of the popularity of these two I/O libraries, the concentration of this research was mainly HDF5 and netCDF. Both these libraries

are built on top of MPI-IO, the parallel I/O feature introduced with MPI-2.

MPI-IO was developed in 1994 in the IBM's Watson Laboratory in order to provide parallel I/O support for MPI, and in 1996 MPI Forum decided to incorporate MPI-IO in MPI-2 standard. The MPI-IO library is a very nicely built interface from the point of view of MPI users. All MPI-IO function calls are very similar to MPI calls and follows the same naming conventions as that of MPI. Writing MPI files is similar to sending MPI messages and reading MPI files is similar to receiving MPI messages. Furthermore MPI-IO fully embraces the versatility and flexibility of MPI data types - and then takes this concept one step further in defining the MPI file views.

Sending and receiving messages can be blocking or non-blocking. For optimizing the parallel program, it may help to use non-blocking communications. The general idea here is that you can start a message send, and then immediately return to computations, while the message is being sent in the background. Similarly you can keep computing while receiving a message in the background. This would work best on systems where there are processors that are dedicated to I/O and do not in general participate in computations. The IBM BlueGene/L is an example of such a machine. In a similar way, MPI-IO also allows writes and reads of files in a normal, i.e., blocking mode, and then in the non-blocking mode – asynchronously; so that computations can be carried out, while the file is being read or written in the background.

Another very important feature supported by MPI-IO is the concept of collective operations. Processes can access MPI files each on its own, or collectively at the same time. The latter allows for read and write optimizations that can be implemented on various levels.

With respect to writing of wave-functions in MFDn to disk, the MPI-IO library can be expected to give much performance improvements in the I/O to disk. However, the files written by MPI-IO are written in raw binary form and are not human readable which is a desired feature for the wave-function file. Another shortcoming of MPI-IO library is that the files are not portable. This means that the files written on one machine may or may not be read correctly on some other machine depending on the architecture. Portability of files is another highly desirable feature when moving forward to I/O libraries from raw one processor

I/O. Files created by several research groups would ideally be required to be usable by any other group, therefore portability becomes an important requirement for I/O libraries.

In spite of the shortcomings of MPI-IO, it is a highly efficient I/O library which performs parallel I/O. There are other I/O libraries available which make use of the highly efficient MPI-IO library as an underlying layer and introduce important features such as portability and human readability of files.

4.3 Network Common Data Form

The Network Common Data Form, or netCDF, is an interface to a library of data access functions for storing and retrieving data in the form of arrays. An array is an n-dimensional (where n is $0, 1, 2, ...$) rectangular structure containing items which all have the same data type (e.g., 8-bit character, 32-bit integer). A scalar (simple single value) is a 0-dimensional array.

NetCDF is an abstraction that supports a view of data as a collection of self-describing, portable objects that can be accessed through a simple interface. Array values may be accessed directly, without knowing details of how the data are stored. Auxiliary information about the data, such as what units are used, may be stored with the data. Generic utilities and application programs can access netCDF datasets and transform, combine, analyze, or display specified fields of the data. The development of such applications has led to improved accessibility of data and improved re-usability of software for array-oriented data management, analysis, and display.

The netCDF software implements an abstract data type, which means that all operations to access and manipulate data in a netCDF dataset must use only the set of functions provided by the interface. The representation of the data is hidden from applications that use the interface, so that the way data is stored could be changed without affecting existing programs. The physical representation of netCDF data is designed to be independent of the computer on which the data were written. The netCDF data format is "self-describing". This means that there is a header which describes the layout of the rest of the file, in particular the data

arrays, as well as arbitrary file metadata in the form of name value attributes. The format is platform independent, with issues such as endianness being addressed in the software libraries. The data arrays are rectangular, not ragged, and stored in a simple and regular fashion that allows efficient sub-setting.

An extension of netCDF for parallel computing called Parallel-NetCDF (or PnetCDF) has been developed by Argonne National Laboratory and Northwestern University. This is built on top of MPI-IO, the I/O extension to MPI communications. Using the high-level netCDF data structures, the PnetCDF libraries can make use of optimizations to efficiently distribute the file read and write applications between multiple processors.

4.4 Hierarchical Data Format

Hierarchical Data Format, commonly abbreviated HDF, HDF4, or HDF5 is the name of a set of file formats and libraries designed to store and organize large amounts of numerical data. Originally developed at the National Center for Supercomputing Applications, it is currently supported by the non-profit HDF Group [15], whose mission is to ensure continued development of HDF5 technologies, and the continued accessibility of data currently stored in HDF.

In keeping with this goal, the HDF format, libraries and associated tools are available under a liberal, BSD-like license for general use. HDF is supported by many commercial and non-commercial software platforms, including Java, Matlab, IDL, and Python. The freely available HDF distribution consists of the library, command-line utilities, test suite source, Java interface, and the Java-based HDF Viewer (HDFView).

There currently exist two major versions of HDF, HDF4 and HDF5, which differ significantly in design and API. We have used the newer version of HDF5 in MFDn for parallelizing the I/O of wave-functions. The HDF5 format is designed to address some of the limitations of the HDF4 library, and to address current and anticipated requirements of modern systems and applications.

HDF5 simplifies the file structure to include only two major types of object:

- Datasets, which are multidimensional arrays of a homogenous type.
- Groups, which are container structures which can hold datasets and other groups.

This results in a truly hierarchical, file-system-like data format. In fact, resources in an HDF5 file are even accessed using the POSIX-like syntax `/path/to/resource`. Metadata is stored in the form of user-defined, named attributes attached to groups and datasets. More complex storage API's representing images and tables can then be built up using datasets, groups and attributes.

In addition to these advances in the file format, HDF5 includes an improved type system, and dataspace objects which represent selections over dataset regions. The API is also object-oriented with respect to datasets, groups, attributes, types, dataspaces and property lists. The latest version of NetCDF, version 4, is based on HDF5.

CHAPTER 5. Performance of MFDn with HDF5

5.1 HDF5 with Collective and Independent I/O

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 [30]. 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 simultaneously. 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.

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. The difference in the operation of Independent I/O and Collective I/O can be seen in Figure 5.1. 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 [30]. 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 [31]. Apart from data layout, I/O performance is also affected by other factors such as caching, network bandwidth, latency, and the file system used.

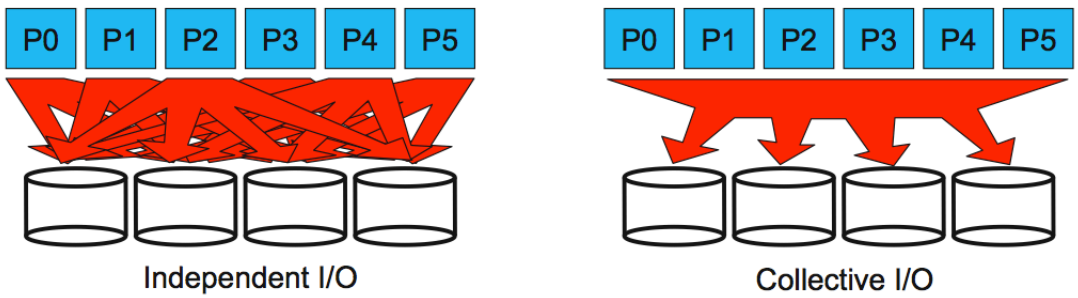

Figure 5.1: Independent vs Collective I/O

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. 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 [30], but in our experience the results obtained were identical with both posix and MPI-IO.

5.2 Integration with MFDn

In the version of MFDn where the wave-function file is written without parallel I/O, 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 disk. At the same time, the other diagonal processors send their share of the wave-functions to the root processor. This process has obvious communication costs associated with it. Moreover the root processor takes all the load of writing to disk. We have successfully parallelized this task with the use of parallel I/O library *Parallel HDF5*. 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 5.2 shows the older sequential write pattern in MFDn and Figure 5.3 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 hyperslabs 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 wrapper functions are: `phdfread` and `phdfwrite`. The interface is straightforward and minimal HDF5 parameters are required while calling the routines for writing and reading the data.

5.3 Creation of the Wave-functions

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, diag-

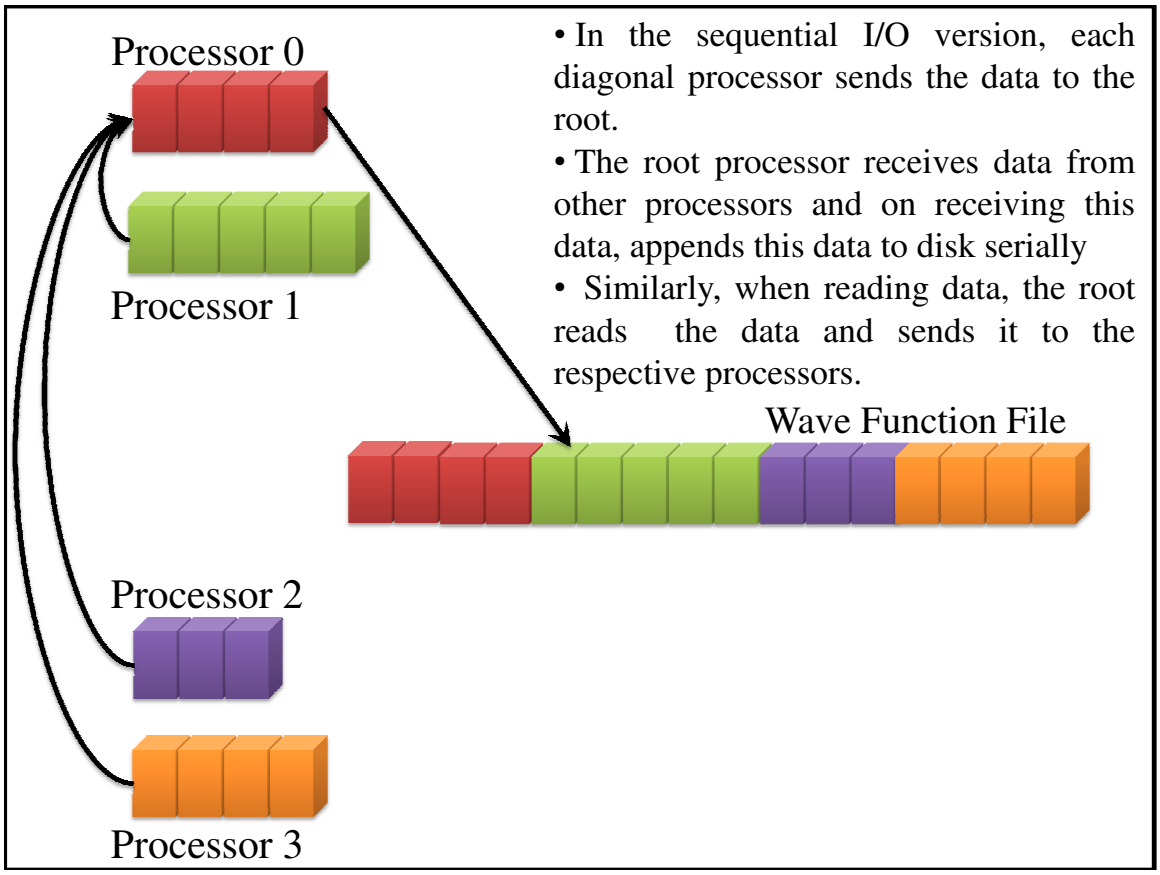

Figure 5.2: Sequential I/O

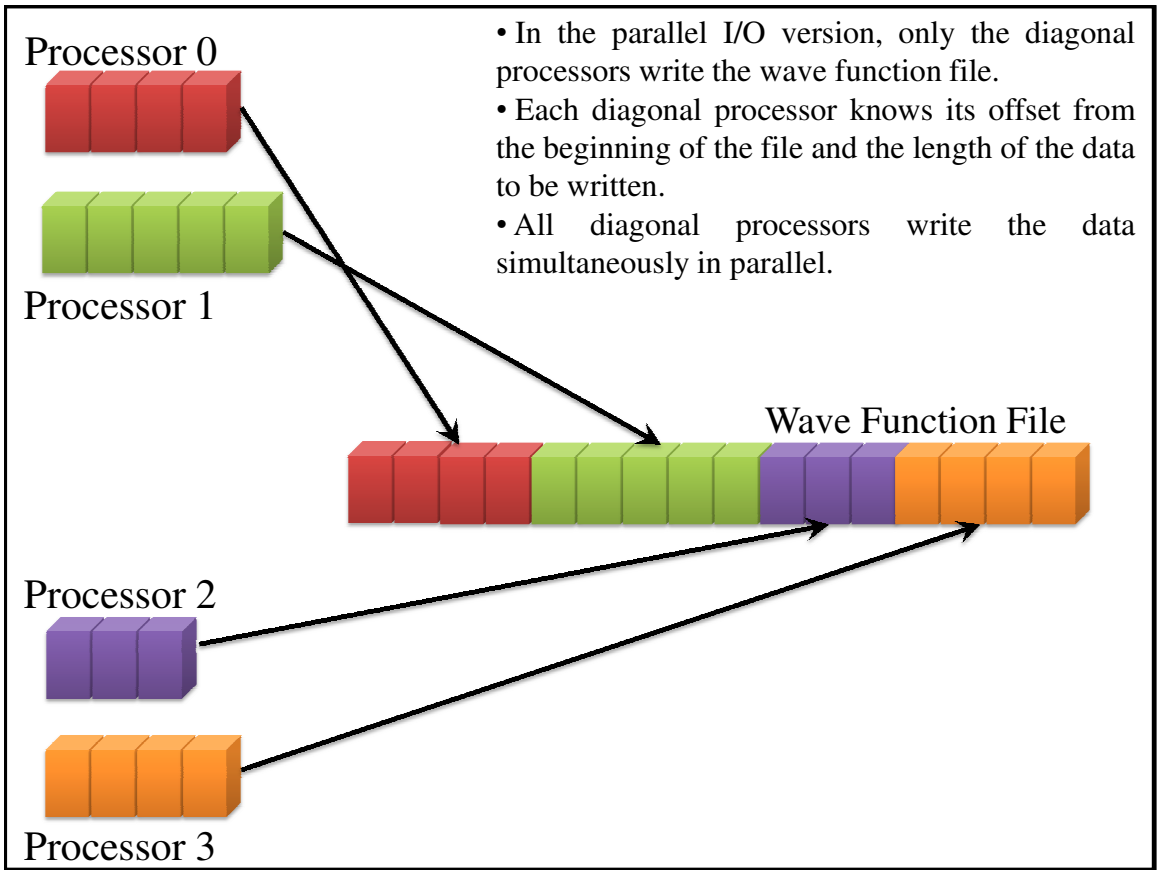

Figure 5.3: Collective I/O

onalizes 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 sub-routines to verify numerical convergence of the diagonalization procedure and to calculate a suite of observables.

5.3.1 The Wave-Function File

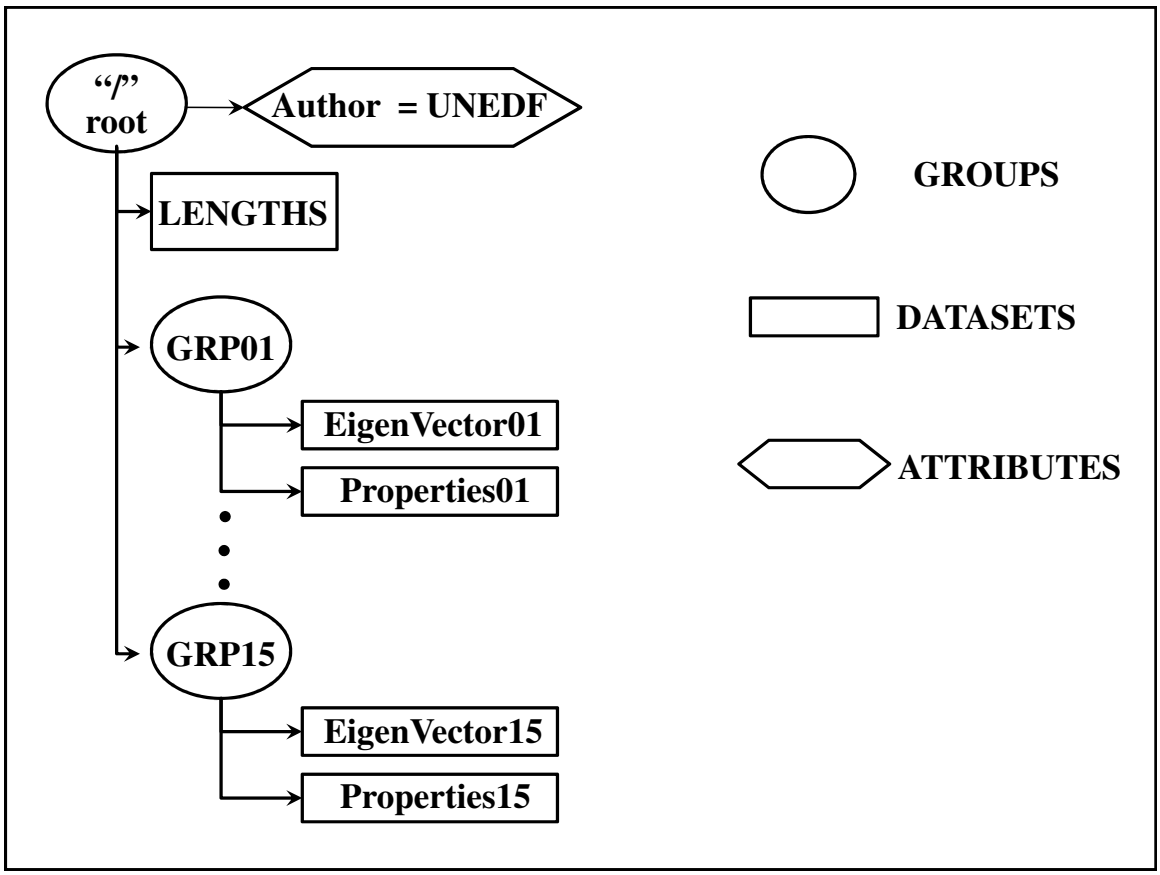

Figure 5.4: The Wave-Function File

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 unix 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 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 5.4 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.

5.4 MFDn IO Simulator

As the experiments grow, the size of the wave-function file as well as the run-time of MFDn continues to grow exponentially. A normal run of MFDn requires $\frac{n(n+1)}{2}$ processors, where n is an odd number and also signifies the number of diagonal processors for the MFDn run. After diagonalization of the Hamiltonian matrix H, 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. For larger runs of MFDn, the number of idle processors at the time of wave-function I/O is rather large, even when using parallel I/O. Since only n processors do the actual I/O out of the $\frac{n(n+1)}{2}$ processors, the remaining processors are idle. This results in a wastage of lot of precious time on supercomputers which are very expensive to use. For example, the supercomputers available on NERSC have a limited amount of time for each group and using most of this time for developing and testing the implementation of HDF5 library with MFDn becomes extremely costly as there are $\mathcal{O}(n^2)$ idle processors during the

I/O of wave-functions. In order to overcome this problem and still 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 binary and HDF5 format input files. Once the raw binary and HDF5 files are available, the I/O performance is measured by a set of programs that simulate posix I/O and HDF5 I/O.

5.5 Performance Results

In this section, we report the I/O performance observed by using the parallel 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 cluster *Franklin* 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 GFlops/sec. Each compute node has 8 GBytes of memory. The parallel file system on *Franklin* is the *Lustre* file system. Figures 5.5– 5.7 show the performance of parallel HDF5 for different wave-function file sizes. The processors on the X-axis represent the n diagonal processors. For same file sizes, the actual MFDn code would run on $\frac{n(n+1)}{2}$ 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 5.8 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

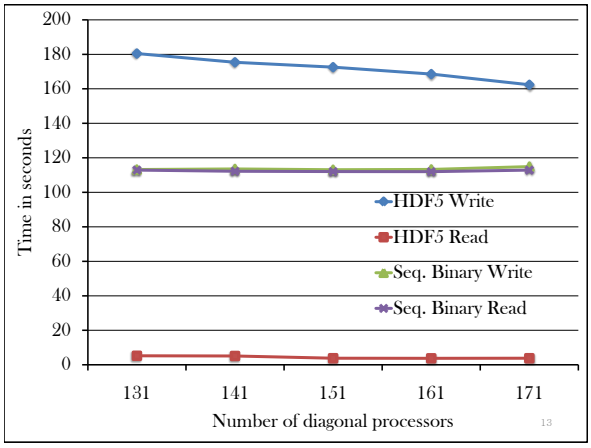

Figure 5.5: I/O Performance for 33 GB File

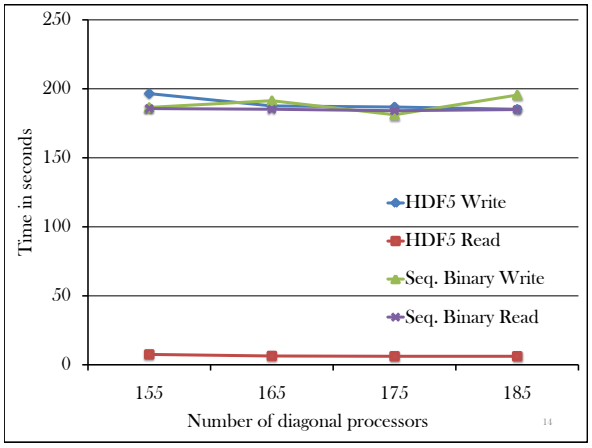

Figure 5.6: I/O Performance for 55 GB File

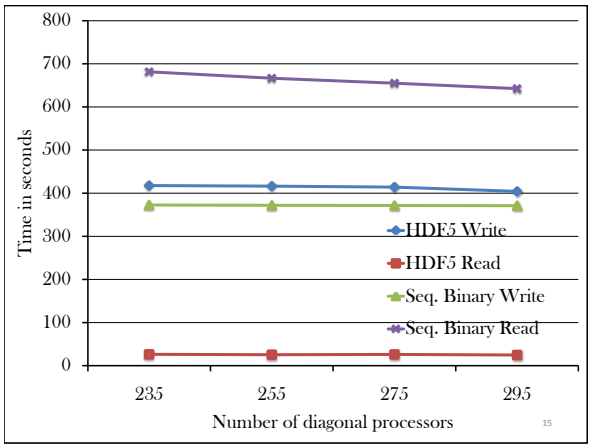

Figure 5.7: I/O Performance for 111 GB File

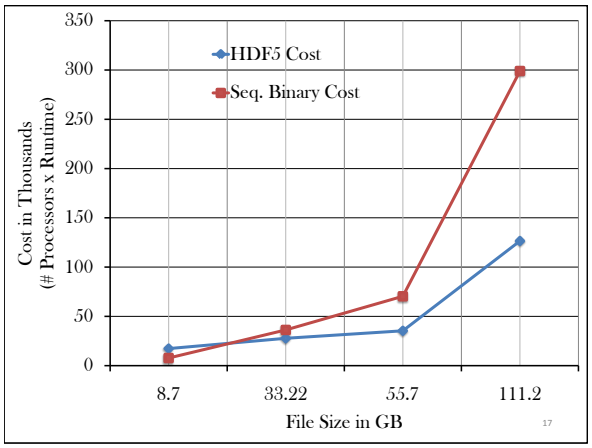

Figure 5.8: Cost of using Parallel I/O

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 size considered here, 111 GB, the difference between parallel HDF5 and sequential binary is approximately a factor of 5 in favor of HDF5. Figure 5.8 shows a cross-over point of HDF5 and sequential binary cost at around 20 GB. An output file of about 20 - 100 GB is typical of our large runs where I/O is a significant factor.

Even though the HDF5 write operations are serialized essentially making the write operations sequential, the performance of HDF5 writes become comparable, and even better as compared to that of the sequential binary writes as the file size and number of processors increase. To analyze this in more detail, we conducted another set of experiments. The testbed

for these new experiments was also the Franklin supercomputer at NERSC. However some significant changes were made to the Franklin supercomputer before the new experiments. These changes are listed below.

- The version of HDF5 had been upgraded from 1.8.0 to 1.8.4.1.
- The version of Lustre had been upgraded from 1.4.12 to 1.6.5.
- New hardware was installed to add bandwidth to Franklin's I/O subsystem and improve metadata operations. It also has the potential to provide additional stability and reducing I/O contention among jobs.
- Compute Nodes were increased from 20 to 60.
- I/O Server Nodes (OSS) for scratch file systems were increased from 20 to 48.
- Object Storage Targets (OSTs) for scratch file systems were increased from 80 to 96.

In the new experiments we measure the sequential binary write times with and without the communication overhead. Figure 5.9 shows the sequential binary and the parallel HDF5 write times. It can be observed that as the file size and the number of processors increase, the gap between sequential binary and parallel HDF5 increases. Despite the fact that HDF5 write operations are serialized making it theoretically equivalent to sequential writes, this gap is due to difference in the access methods. HDF5 allows direct access to parts of the file without first parsing the entire contents. Whereas in case of sequential binary, the entire contents need to accessed before reaching to the point where the next data has to be written. In addition to this we can see that for larger file sizes, the plots for sequential binary with and without communication begin to split up. This split is expected to be more for larger file sizes and more processors because, as the file size increases, the amount of data at each processor increases. Sending larger amounts of data to the root processor incurs significant communication overhead in case of sequential binary write operations. In addition to the large data, as the number of diagonal processors increase, the communication overhead increases significantly as more processors need to exchange data with the root processor. In parallel

HDF5, although the write operations are serialized, there is no communication required as each processor writes the data to disk itself, thereby eliminating communication overhead. This means that despite locking infrastructure and the slow HDF5 write operations, the cost of using HDF5 is less as compared to sequential binary I/O for large datasets. According to the new experiments, parallel HDF5 outperforms sequential binary at a threshold lesser than that in the previous experiments of Figures 5.5-5.8. This can be accredited to the upgrade of the Lustre file system, parallel HDF5, and considerable hardware and software changes on the Franklin supercomputer.

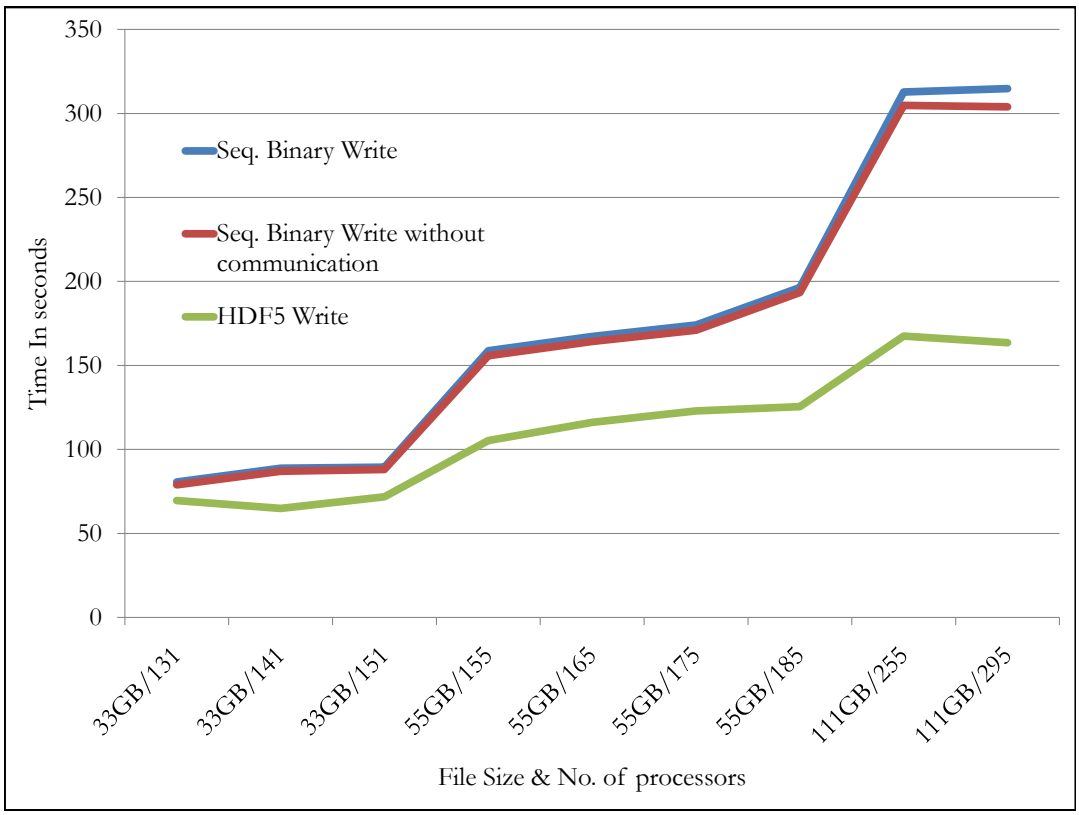

Figure 5.9: Comparison of Sequential Binary and Parallel HDF5 Write Operations

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 [32], 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 data 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* file system 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" by 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 read operations. 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.

In conclusion, 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 lesser compared to sequential I/O for sufficiently large datasets. This work shows that parallel I/O libraries can be of great value to scientific applications dealing with large datasets. The investigation of the difference between collective and independent I/O and situations in which they are effective are much useful for the scientific community.

CHAPTER 6. Performance of MFDn as integrated component

Chapter 5 showed the improvement in performance of MFDn as a standalone application by using parallel I/O libraries. The use of parallel I/O shows significant performance improvement for file sizes above 20 GB which are typical sizes for normal MFDn runs. For ab-initio calculations, often MFDn is not the only code which needs to be run. MFDn runs along with several other programs to get nuclear physics results. These codes are categorized as upstream and downstream codes. Upstream codes are those which run before MFDn and provide suitable input parameters for MFDn. Downstream codes are those which use the output of MFDn and do some other post processing to obtain certain physical observables.

6.1 Need of Checkpointing

In addition to the importance and need of checkpointing as discussed in Section 1.3, there is one additional hurdle which makes checkpointing all the more desirable for MFDn. There are upper bounds on allowed batch time on supercomputers where the MFDn related scripts are submitted and run. On NERSC supercomputers, this limit is 48 hours, which is surely not enough to find the global or even local minimum for expensive function evaluations as described in this thesis. Hence, checkpointing is a feature which is extremely valuable and would be utilized as a routine procedure to restart the integrated code for the next maximum time allowed by the queuing system. This integrated code would involve heavier nuclei, larger N_{max} and N_{shell} which runs for a very long time as compared to the time limits on batch jobs. Therefore, it is imperative that the application history (snapshot) is saved and is re-used the next time the subsequent batch job is started. Hence checkpointing is an important feature which is highly desirable in MFDn for more than one reason.

6.2 Integration example of MFDn and Optimization codes

Figure 6.1 shows the integration of MFDn with the optimization codes. The codes that run before MFDn, namely `Heff, M2, GOSC` are called the upstream codes and they prepare the input files for MFDn. The Newuoa/VTDirect search algorithms use some input parameters (a vector $x \in \mathbb{R}^n$) and the evaluation of the function $F(x)$ as required by Newuoa is generated by running MFDn inside Newuoa. This function value needs to be minimized which requires searching for the appropriate input parameters. The search algorithm optimizes the function value to find the local or global minima. This requires searching for the input parameters within some boundary range and using efficient algorithms to make this process of minimizing the function value as fast as possible. It is clear that this process involves running of MFDn for each iteration of Newuoa/VTDirect until the search algorithm succeeds in finding a local or global minima.

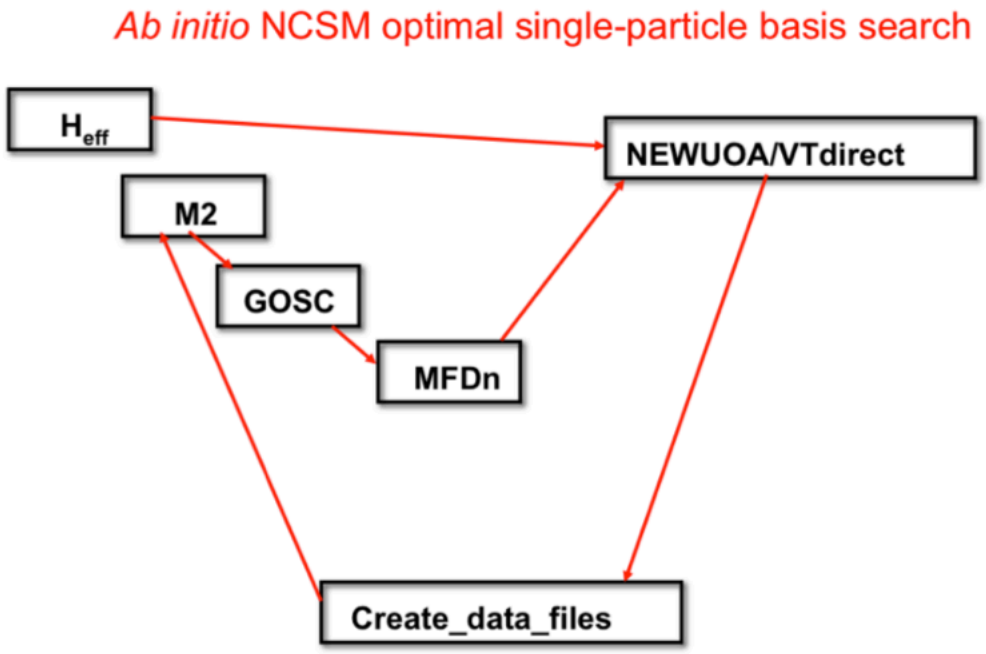

Figure 6.1: Components of the ab-initio calculations

6.2.1 Purpose of the Integration

There are several programs that take part in the integrated design of the script. All these codes are created such that they use the output of other programs and generate the input for following codes. Thus, it is beneficial to integrate all the upstream codes, MFDn and search algorithm as a single component. This integration is done by using a unix shell script. The different components of this script are:

1. `Heff`: Program to input 2-body RCM matrix elements of H and generate Heff (or Veff) in chosen subspace, the P-space, using the method of Wilson-Lee-Suzuki-Okamoto-Okubo, as presented in [4, 9].

2. `m2v9`: This code takes as input, the translationally invariant full NN-potential matrix elements and transforms them to the sp basis.

3. `gosc`: Program for rearranging the No-Core Matrix Element files generated by `m2v9` to formats convenient to use.

4. `MFDn`: Nuclear physics tool that uses the input files prepared by the upstream codes along with other input files such as interaction files and calculates observables such as wave-functions and ground state energies.

5. `Newuoa`: Searches for optimal parameters for MFDn to minimize the objective function value (ground state energy for certain nucleus) that is one of the outputs of MFDn.

Since all the above programs seem to use the outputs of other programs as input files and generate output which is usable by other programs, an integrated system was desirable which can run all the above codes seamlessly without user intervention.

6.3 What is Newuoa

An important component of the integrated design described above is Newuoa. The name of the software, NEWUOA stands for NEW Unconstrained Optimization Algorithm. It is a

development of UOBYQA [33], which is unsuitable for large number of variables n, because the quadratic models of UOBYQA are constructed by interpolation to $\frac{1}{2}(n+1)(n+2)$ values of the objective function. On the other hand, the number of interpolation conditions in NEWUOA is a parameter, which reduces the magnitude of the routine work of each iteration from n^4 to between n^2 and n^3. Thus NEWUOA has solved problems successfully with up to 200 variables, which is rather onerous for UOBYQA.

Complete details of NEWUOA software and the algorithm can be found in [28] and [33].

6.4 Implementation of Checkpointing and Interface with Newuoa

As described before, checkpointing is a technique for inserting fault tolerance into computing systems by saving a snapshot of the current application state, and later on, use it for restarting the execution. For checkpointing, it is important to identify what information needs to be saved and what information does not need to be saved. Checkpoints are taken in anticipation of the potential need to restart a software process.

Many ordinary batch processes on supercomputers are time-consuming, as are backup and restore operations. They consist of many units of work. If checkpointing is enabled, checkpoints are initiated at specified intervals, in terms of units of work or of processing time. At each checkpoint, intermediate results and a log recording the process's progress are saved to non-volatile storage. The contents of the program's memory area may also be saved.

The purpose of checkpointing is to minimize the amount of time and effort wasted when a long software process is interrupted by a hardware failure, a software failure, or resource unavailability. With checkpointing, the process can be restarted from the latest checkpoint rather than from the beginning. Checkpoints should occur frequently enough to minimize wasted effort when a restart is necessary but not so frequently as to prolong the process unduly with checkpoint overhead. Optimal checkpoint frequency depends on the mean time between failures (MTBF), among other factors.

Before mentioning how checkpointing has been implemented in Newuoa, we make a brief explanation of the input parameters of Newuoa. The input parameters for Newuoa include

the number of variables n, the maximum number of function evaluations, the upper and lower bounds of the parameters. We have introduced 1 more parameter for specifying the mode of Newuoa which could be one of 3 modes.

This additional parameter is defined as the `RESTART` switch which defines the mode of Newuoa. This mode can be one of the 3 values as described below:

- OFF (0): Newuoa runs without checkpointing i.e. No history is saved and therefore, restarts are not possible.
- SAVING (1): Checkpoint file is saved at frequent intervals. This file is updated regularly with the function evaluations that have been completed along with the values of the parameters.
- RECOVERY (2): Newuoa starts in restart mode and uses the checkpoint file to run through the evaluations that have been completed before. Once all the evaluations in the checkpoint file have been read, Newuoa continues its normal execution. In addition to recovery, this mode internally also implements the *Saving* mode in order to facilitate multiple restarts.

The work in this thesis adopts a User Level and Non-Transparent checkpointing method that records or recovers function evaluation logs via file I/O. [17] categorizes such a method as "user level" and "nontransparent" because the checkpointing implementation is visible in the source code and is implemented outside the operating system without using any system level utilities. It requires more programming effort than simply applying system level transparent tools. This method of checkpointing is flexible and precise in choosing what information to save, instead of dumping all the relevant program and even system data. In this work, the data needed to recover previously completed function evaluations are chosen as the checkpointing state information. The checkpointing switch RESTART can be 0 ("off"), 1 ("saving"), or 2 ("recovery"). Checkpointing errors are mainly related to the file under consideration. This could be in the process of opening, reading, writing, or finding checkpoint logs. For "saving", the program will report an opening error if the default checkpoint file already exists, in order

to prevent the saved checkpoint logs from being overwritten. Hence, an old checkpoint file should be either removed or renamed before starting another "saving" run. The opening error also occurs when "recovery" cannot find the needed checkpoint file. Note that the checkpoint file has a fixed name (`newuoachk.dat`). The metadata information about the total number of completed function evaluations is saved in another file called `nfev_comp_file`. Changing any of the input parameters of Newuoa will result in different values of the function evaluation. However, a parameter defined for the maximum number of function evaluations can be changed to delay or expedite the stopping condition on NEWUOA in case of restart runs.

A checkpoint log consists of the current iteration number t, a vector x of function variables, and the function value $F(x)$ at x. The "saving" run records each evaluation as a checkpoint log in the file. For "restart" runs, assuming the computer architecture is the same, the functions are evaluated in the same sequence for the number of function evaluations as saved in the `nfev_comp_file`. This is because NEWUOA is a deterministic algorithm, i.e. given the same input, it will produce the same output. This deterministic property of NEWUOA plays a very important role in the implementation of checkpointing. Therefore, the "recovery" run loads all the checkpoint logs, or those that are within the iteration limit if specified. These logs are stored in a list in the same order as in the file, and will be recovered in that order as the program progresses.

Fault-tolerant checkpointing is very important from the point of view of the users of this feature. This is because, in case of hardware or power failure, if the checkpoint file gets corrupted, then the users should have a way to detect such corruption. Guaranteeing foolproof checkpointing and checkpoint files can enable the users to look for other corrupted data related to the application in case of a power or hardware failure. Since power failures are unpredictable, uncorrupted checkpoint files can be guaranteed by having two checkpoint files. In such a case, even if there is a power failure while one of the checkpoint file is being updated, the other checkpoint file is not open at that time and the data in it is guaranteed to be valid.

Checkpointing is a valuable feature of any high performance computing application due to fact that these applications generally run for many hours and even days at a time. An

important need for this feature has been discussed before in this thesis owing to the fact that MFDn search codes need to be stopped and run many time which warrants a need for checkpointing. Second need of this feature come from the fact that supercomputers with batch scheduling typically have an upper bound on the time any job is allowed to execute. Due to the clear need for checkpointing in general in high performance computing systems, it is imperative that this feature should be implemented at as many levels of application as possible because the expense of doing checkpoint/restart is negligible as compared to the actual work needed in most high performance computing applications. Checkpointing is also an operating system component that provides checkpoint and restart services to processes. The application, in most cases, remains unaware that there was any interruption to its execution. The same feature is implemented at a higher level in scientific computing applications. These applications may be divided into two types: serial applications and parallel applications. For the purposes of this thesis, a serial application is an application that executes under a single instance of an Unix kernel. Creating a valid context fileset for a serial application require no coordination with processes running on remote nodes.

The checkpointing feature implemented in NEWUOA serial code is an example demonstrating the importance and utility of checkpointing in the field of computer science and high performance scientific applications. The parallel applications are written using an MPI (Message Passing Interface) library. Checkpointing these applications requires modifications to the MPI library. The MPI library must be modified to coordinate the execution of an application so that a consistent distributed checkpoint may be taken [34]. Checkpointing can be further implemented in MFDn at various levels. The complete nuclear physics application suite that is used for obtaining certain observables is a collection of serial and parallel codes as previously seen in this thesis. Implementing application level checkpointing in the upstream codes that run before MFDn and in MFDn itself will provide a much greater flexibility in restarting the batch jobs at certain points currently not feasible. A finer granularity of checkpointing implementation will provide a much greater flexibility in restarting jobs at checkpoints which are not currently implemented. Thus greater amount of work already done will not need to be

re-run.

6.5 Overhead Evaluation: Newuoa

The following experiments measure the checkpointing overhead in NEWUOA. The evaluation limit of 150 is used for the three test cases. Table 6.1 reports the time of a single function evaluation under different cases: T_{fe}, and the time of a single function evaluation in case of restarts. This time incorporates the time for "saving" T_{sv}, and the time for "recovery" T_r. First, note that the sum of T_{sv} and T_r is of the order of micro-seconds. This means the save-recovery overhead is tiny even for the cheap functions. Secondly, the saving overhead depends heavily on the number of iterations, because the checkpoint logs are flushed to the file at the end of each iteration. The average saving overhead per iteration is approximately 400 μ seconds. For real-world applications, the total function evaluation time T_{fe} is usually much greater (e.g., $T_{fe} \approx 1000$ for the cases in Table 6.1). In such cases, the saving overhead would be negligible compared to the cost of function evaluations. In summary, checkpointing overhead is insignificant compared to the benefit of saved computation for expensive function evaluations.

Table 6.1: Checkpointing overhead for three test functions. T_{fe} is the time of a single function evaluation, T_{sv} is the time to save checkpoint data, and T_r is the time for recovery.

Nucleus, N_{max}, N_{shell}	T_{fe} (seconds)	$T_{sv} + T_r$ (μ seconds)
$^4He, 6, 14$	837	778
$^4He, 8, 14$	880	777
$^4He, 10, 14$	1100	800

CHAPTER 7. Conclusions

7.1 Summary and Future Work

This body of work has proposed a design and implementation of the integration of the MFDn and parallel I/O library. The use of simple file I/O has been extended in the implementation of checkpointing. We have identified sequential I/O as a bottleneck in the MFDn code performing nuclear structure calculations. Identifying this bottleneck helped in understanding the utility of file I/O in high performance applications. Parallel I/O leads to significant performance improvement in applications where the data to be accessed on disk is very large. By distributing the workload of a single processor to several processors leads to much higher I/O speed in addition to other advantages such as portability and human readability of files. 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. There are two reasons for this: (1) Direct access to any part of the HDF5 file and (2) No communication overhead. The utility of using file I/O for improving the performance of scientific applications has been further extended by implementing checkpointing in the optimization code NEWUOA. The implementation of checkpointing in Newuoa is only an example and the same can be extended to the other programs in the nuclear physics application suite. We have shown that the overhead of using this feature is negligible as compared to the amount of work that can be recovered by using very small checkpoint files. The checkpoint files enable computationally intensive applications to store a necessary amount of history in a file using which the application can later be restarted. This restart is transparent from the point of view of the application.

Our contribution is to show how simple I/O as well as a relatively complicated parallel I/O libraries can be of great value to scientific applications dealing with large datasets and large runtimes. Parallel I/O is of importance in the load balancing, performance improvement, portability and human readability of large datasets. Checkpointing related file I/O is of importance in applications dealing with runtimes ranging from a few hours to several days as it provides a resume capability to computationally intensive jobs thus preventing the potential loss of large amount of previously done work.

Much work can be done to further improve the performance of nuclear physics calculations. We have investigated the difference between collective and independent I/O and situations in which they are effective. Different data layouts and architectures affect the performance of parallel I/O and dictate the mode of I/O to be used. With these results in hand, parallel I/O can be further implemented for other large datasets in use by MFDn. Furthermore, taking into consideration the different hardware and software libraries used while optimizing the I/O performance of MFDn, it would be more beneficial and conclusive to model the performance of the available resources. This is so, because even though the performance of parallel HDF5 is much better than sequential binary for large datasets, it would be helpful to know if this performance can be further improved. In general, by investing more time to study the architecture of the machines, the number of I/O forwarding nodes and network architecture, computer scientists and developers can have an idea of the possible optimal performance beforehand.

Checkpointing in NEWUOA can also be further extended to other codes that are a part of the nuclear physics application suite doing ab-initio calculations. Implementing checkpointing in MFDn as well as other upstream and downstream codes will provide higher granularity and flexibility in the restarting of MFDn batch jobs. The simple file I/O of checkpointing can also be converted to HDF5 format thus providing the features such as portability and human readability to checkpoint files.

BIBLIOGRAPHY

[1] S. C. Pieper, V. R. Pandharipande, R. B. Wiringa, and J. Carlson, Realistic models of pion-exchange three-nucleon interactions. *Phys. Rev. C64, 014001, 2001*

[2] G. Hagen, T. Papenbrock, D. J. Dean, and M. Hjorth-Jensen, Medium-mass nuclei from chiral nucleon-nucleon interactions. *Phys. Rev. Lett. 101, 092502, 2008.*

[3] P. Navratil, V. G. Gueorguiev, J. P. Vary, W. E. Ormand, and A. Nogga, Structure of A=10-13 nuclei with two- plus three-nucleon interactions from chiral effective field theory. *Phys. Rev. Lett. 99, 042501, 2007.*

[4] P. Navratil, J. P. Vary, and B. R. Barrett, Properties of 12C in the abinitio nuclear shell-model. *Phys. Rev. Lett. 84, 57285731, 2000.*

[5] S. K. Bogner, et al., Convergence in the no-core shell model with low-momentum two-nucleon interactions. *Nucl. Phys. A801, 2142, 2008.*

[6] P. Maris, J. P. Vary, and A. M. Shirokov, Ab initio no-core full configuration calculations of light nuclei. *Phys. Rev. C79, 014308, 2009.*

[7] P. Maris, A. M. Shirokov, and J. P. Vary, Ab initio nuclear structure simulations: the speculative 14F nucleus. *arXiv:0911.2281.*

[8] J. P. Vary and D. C. Zheng, (unpublished), The Many-Fermion Dynamics Shell- Model Code. Iowa State University,1992

[9] P. Navratil, J. P. Vary, and B. R. Barrett, Large-basis ab-initio No-core Shell Model and its application to 12C. *Phys. Rev. Volume C 62, 054311, 2000.*

[10] P. Sternberg, E. G. Ng, C. Yang, P. Maris, J. P. Vary, M. Sosonkina, and H. V. Le, Accelerating configuration interaction calculations for nuclear structure. *In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Austin, Texas, November 15 - 21, 2008). Conference on High Performance Networking and Computing. IEEE Press, Piscataway, NJ, 1-12. DOI= http://doi.acm.org/10.1145/1413370.141338.*

[11] M. Sosonkina, A. Sharda, A. Negoita, and J. P. Vary, Integration of ab initio nuclear physics calculations with optimization techniques. *Computational Science - ICCS 2008, 8th International Conference, Krakow, Poland, June 23-25, 2008, Proceedings, Part I, 2008,* pp. 833842.

[12] J. P. Vary, P. Maris, E. Ng, C. Yang, and M. Sosonkina, Ab initio nuclear structure - the large sparse matrix eigenvalue problem. *J. Phys. Conf. Ser. 180 (2009) 012083. arXiv:0907.0209, doi:10.1088/1742-6596/180/1/012083.*

[13] MPI-2: Extensions to the Message-Passing Interface, *Message Passing Interface Forum.* pp. 205-264.

[14] MPI-2: Extensions to the Message-Passing Interface, *Message Passing Interface Forum.*

[15] The HDF Group, *http://www.hdfgroup.org*

[16] The NetCDF Homepage, *http://www.unidata.ucar.edu/software/netcdf/*

[17] J. S. Plank, An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. *Technical Report UT-CS-97-372, University of Tennessee, Knoxville, Tennessee, 1997.*

[18] T. G. Kolda, R. M. Lewis, and V. Torczon, Optimization by direct search: new perspectives on some classical and modern methods. *SIAM Review. SIAM Review,vol. 45,* pp. 385482, 2003.

[19] P. M. Pardalos, H. E. Romeijn, and H. Tuy, Recent developments and trends in global optimization. *Journal of Computational and Applied Mathematics,vol. 124,* pp. 209228, 2000.

[20] R. B. Schnabel, A view of the limitations, opportunities, and challenges in parallel nonlinear optimization. *Parallel Computing, vol. 21,* pp. 875905, 1995.

[21] A. Migdalas, G. Toraldo, and V. Kumar, Nonlinear optimization and parallel computing. *Parallel Computing, Volume 29,* pp. 375391, 2003.

[22] U. M. Garcia-Palomares and J. F. Rodriguez, New sequential and parallel derivative-free algorithms for unconstrained minimization. *SIAM Journal on Optimization, vol. 13,* pp. 7996, 2002.

[23] T. Weise. Global Optimization Algorithms - Theory and Applications. *Version: 2008-09-24*. http://www.it-weise.de/

[24] M. P. Wachowiak. High Performance and Parallel Optimization. Department of Computer Science and Mathematics, Nipissing University.

[25] D. R. Jones, C. D. Perttunen, and B. E. Stuckman, Lipschtzian optimization without the Lipscitz constant. *Optimization Theory and Applications, Volume 79,* pp. 157-181.

[26] J. He, L. T. Watson, and M. Sosonkina, Algorithm XXX: VTDIRECT95: Serial and Parallel Codes for the Global Optimization Algorithm DIRECT. *Association for Computing Machinery, Inc.*

[27] M. J. Box, A New Method of Constraint Optimization and a Comparison with Other Methods. *Computer Journal. 8 (1965),* pp. 42-52.

[28] M. J. D. Powell, The NEWUOA software for unconstrained optimization without derivatives. *Large-Scale Nonlinear Optimization, Volume 83, pp. 255-297, 2006.*

[29] R. Latham, The Parallel-netCDF I/O Library. *Argonne National Laboratory, Jan 2010.*

[30] M. Yang and Q. Koziol, Parallel HDF5 Hints. *NCSA HDF group*

[31] C. M. Chilan, M. Yang, A. Cheng, L. Arber, Parallel I/O Performance Study With HDF5, A Scientific Data Package. *TeraGrid 2006: Advancing Scientific Discovery, June 12-15, 2006.*

[32] HDF5 USERS, *http://www.hdfgroup.org/HDF5/users5.html*

[33] M. J. D. Powell, UOBYQA: Unconstrained Optimization by Quadratic Approximation. *Math. Prog. B, Vol. 92, pp. 555-582, 2002.*

[34] J. Duell, P. Hargrove, and E. Roman, Requirements for Linux Checkpoint/Restart. *LBNL-49659 Checkpoint Requirements-V1-8, May 2, 2002.*