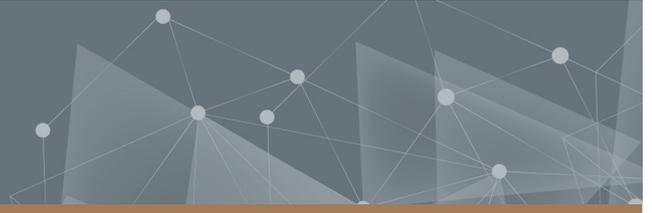




CHALMERS
UNIVERSITY OF TECHNOLOGY



Parallelization of computational tools for the no core shell model

A computational study including statistical ab initio predictions
for the ^8Be decay energy threshold

Master's thesis in Physics

JOHANNES HANSSON

DEPARTMENT OF PHYSICS

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022
www.chalmers.se

MASTER'S THESIS 2022

Parallelization of computational tools for the no core shell model

A computational study including statistical ab initio predictions for the ${}^8\text{Be}$ decay energy threshold

Johannes Hansson



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Division of Subatomic, High Energy and Plasma Physics
Theoretical Subatomic Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022

Parallelization of computational tools for the no core shell model
A computational study including statistical ab initio predictions for the ^8Be decay energy threshold
Johannes Hansson

© Johannes Hansson, 2022.

Supervisor and examiner: Christian Forssén, Department of Physics

Master's Thesis 2022
Department of Physics
Division of Subatomic, High Energy and Plasma Physics
Theoretical Subatomic Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone: +46 (0)31-772 1000

Typeset in L^AT_EX
Printed in Sweden by Chalmers digitaltryck
Gothenburg, Sweden 2022

Parallelization of computational tools for the no core shell model

A computational study including statistical ab initio predictions for the ${}^8\text{Be}$ decay energy threshold

Johannes Hansson

Department of Physics

Chalmers University of Technology

Abstract

The many-body Schrödinger equation is of fundamental importance in nuclear physics. It can be (approximately) solved by employing the no core shell model (NCSM). Using this numerical method in nuclear simulations is a computationally heavy task that requires powerful computer hardware and purpose-built software. In this project we use the NCSM code JupiterNCSM to study an eight-nucleon system, ${}^8\text{Be}$. This research code has previously only been used for nuclei with $A \leq 6$ nucleons, and our extension to study an eight-nucleon system represents a significant increase in computational requirements. Therefore, code performance is a key factor. We find that a single computer node running JupiterNCSM has insufficient computational power to simulate ${}^8\text{Be}$ at large model spaces. To overcome this limitation we introduce distributed-memory parallelization to JupiterNCSM using the Message Passing Interface, which makes it possible to use the computational capabilities of several computer nodes at the same time. In addition, we optimize other areas of the code, such as data access patterns, to further increase performance. As a result we have managed to extend the predictive reach of the JupiterNCSM software, enabling the study of atomic nuclei in larger model spaces. With the memory architecture and code optimization improvements in place, we then use JupiterNCSM to sample the posterior predictive distribution (PPD) of the decay energy threshold of ${}^8\text{Be}$. We find that our NCSM computations are not fully converged, which leads to low-precision predictions manifested by a wide PPD. Still, the predictions are accurate since they reproduce the experimentally measured decay energy. Future efforts to reach even larger model spaces and achieve better convergence are suggested by identifying the most relevant areas for further improvement.

Keywords: nuclear physics, no core shell model, Lanczos algorithm, large-scale matrix diagonalization, parallelization, high performance computing, MPI

Acknowledgments

I would like to express my deepest gratitude to my supervisor Prof. Christian Forssén for all the guidance and support throughout this project. I am also very grateful to Dr. Tor Djärv for writing the JupiterNCSM software and for helping me use it. Furthermore, I would also like to express my gratitude to Dr. Håkan T. Johansson for interesting and engaging discussions about programming and computer systems. Special thanks should also go to Prof. Andreas Ekström for his encouragement, valuable insights and help with data generation. The computations and data handling were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at National Supercomputer Centre (NSC) partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

Last but not least, I would like to thank my parents Örjan and Maricel, and also my brother Niklas. You are always there to lighten my burdens and share my joy. I would never have come this far without your everlasting support.

Johannes Hansson, Gothenburg, June 2022

Contents

I	Background	1
1	Introduction	3
1.1	The nuclear many-body problem	3
2	The no core shell model	5
2.1	The no core shell model basis	5
2.2	The Lanczos algorithm	7
2.3	Computation of large matrix-vector products in the Lanczos algorithm	8
2.4	JupiterNCSM	10
3	Parallel computing and computer memory	11
3.1	Outline of computational difficulties	11
3.2	Description of a typical high performance computing cluster	12
3.3	Memory in high performance computing systems	12
3.4	Methods of parallelization	13
3.5	OpenMP	14
3.6	The Message Passing Interface, MPI	14
3.7	Hybrid parallelization: OpenMP and MPI	14
3.8	Hybrid parallelization in JupiterNCSM	15
II	Implementation	17
4	Profiling	19
4.1	Profiling single instructions	19
4.2	Study of number of files and file sizes	20
5	Work order for the matrix-vector product	23
5.1	Improved work order	23
5.2	File division between nodes using Venus	24
5.3	Disk load estimation program	25
6	MPI parallelization	27
6.1	From partial sums to JupiterNCSM instructions	27
6.2	MPI implementation details	29
6.3	Checkpointing	29

III	Results	33
7	Computational performance	35
7.1	Feasibility of ^8Be simulations at larger model spaces	35
7.2	Typical run times for the Lanczos solver	38
7.3	Performance profiling of the Lanczos solver Bacchus	40
7.4	Optimized work order	40
7.5	MPI performance scaling	44
8	Statistical study of the ^8Be decay energy threshold	47
8.1	Validation of ^8Be results	47
8.2	^8Be convergence study	48
8.3	Three nucleon force uncertainty study	48
IV	Summary, discussion and outlook	55
9	Summary	57
10	Discussion and outlook	59
10.1	Suggested improvements	60

Acronyms

Acronyms used in this thesis; listed in alphabetical order.

2NF	Two-nucleon forces.
3NF	Three-nucleon forces.
4NF	Four-nucleon forces.
API	Application Programming Interface.
CPU	Central Processing Unit.
EFT	Effective Field Theory.
HDD	Hard Disk Drive.
HPC	High Performance Computing.
I/O	Input/Output.
IOPS	Input/Output operations Per Second.
LEC	Low-Energy Constant.
NVMe	Non-Volatile Memory Express.
PPD	Posterior Predictive Distribution.
QCD	Quantum Chromodynamics.
RAID	Redundant Array of Independent Disks.
RAM	Random Access Memory.
SSD	Solid State Drive.
TSP	Traveling Salesman Problem.

Part I

Background

Chapter 1

Introduction

Nuclear physics is the study of the building blocks of matter and the fundamental forces that govern them. One task in this area of research is to simulate atomic nuclei using realistic models of the nuclear interaction. This task can be achieved by solving the many-body Schrödinger equation with all nucleons as active degrees of freedom. Several computational many-body methods have been developed to approximately solve this problem. One such method is the no core shell model (NCSM) [1]. The NCSM method turns the many-body Schrödinger equation into a large matrix eigenvalue problem. However, the matrix can become very large such that special memory management techniques are required for efficient handling by currently available computers.

This project aims to further develop on an existing research code, JupiterNCSM [2], employing the NCSM method for simulating atomic nuclei. A main task is to introduce distributed-memory algorithms to the current implementation. Another goal is to improve utilization of memory locality to reduce the load on secondary memory, which has been identified as a bottleneck. These improvements would enable the code to fully utilize the computational power of modern computer clusters to speed up calculations. Study of more complex many-body systems would then become possible. With these improved capabilities implemented, this project aims to study a comparatively complex many-body system of eight nucleons, the ^8Be isotope.

One of the limitations of this project is that we will assume that the realistic models of nuclear interactions are precomputed and available for use. We will not construct these models ourselves. Another limitation is that we will only consider a subset of the NCSM simulation toolchain in JupiterNCSM. We will mainly focus on the Lanczos solver Bacchus.

1.1 The nuclear many-body problem

The atomic nucleus is a self-bound quantum-mechanical system composed of one or more strongly interacting nucleons. To study it, we generally refer to the many-body Schrödinger equation

$$\hat{H}|\Psi\rangle = E|\Psi\rangle, \quad (1.1)$$

where E is the energy and $|\Psi\rangle$ is the state of the system. The nuclear Hamiltonian \hat{H} is given by [2]

$$\hat{H} = \hat{T}_{\text{int}} + \hat{V}_{2\text{NF}} + \hat{V}_{3\text{NF}}, \quad (1.2)$$

where \hat{T}_{int} is the intrinsic kinetic energy of the system and $\hat{V}_{2\text{NF}}$ and $\hat{V}_{3\text{NF}}$ represent two- and three-nucleon potentials, respectively. In modern nuclear theory, these potentials come from

chiral effective field theory (χ EFT), a type of effective field theory [3, 4]. EFTs are a class of approximations that simplify physical theories by integrating out effects of short-range physics below the relevant resolution scale [5]. This can, equivalently, be seen as regulating effects of high-energy physics.

The nucleus is held together by the strong nuclear force, which is described by quantum chromodynamics (QCD). This theory is, unfortunately, nonperturbative at the comparatively low energies relevant to nuclear physics [3]. Direct use of QCD is then difficult for describing interactions between nucleons. This is where χ EFT is used to apply controlled approximations.

An important property of χ EFT is that it predicts the appearance of many-nucleon forces. These forces cause interactions between two or more nucleons, and are referred to as two-nucleon forces (2NF), three-nucleon forces (3NF), etc. The inclusion of 3NFs increases the accuracy of the description, but also adds significant complexity to the model. This project uses theory and computational tools for nuclear interactions including 2NFs and 3NFs. Using four-nucleon forces (4NFs) in the calculations would not improve accuracy significantly, but it would increase computational requirements drastically. The inclusion of 4NFs is therefore outside the scope of this project [1, 3].

In the χ EFT framework, it is possible to parametrize the contributions of these many-nucleon forces, as is summarized in equation (1.2) where we have three terms: a kinetic term, a term representing 2NFs and a term representing 3NFs. From χ EFT we know that we can further split the interactions into a linear combination of contributions, scaled by χ EFT parameters known as low-energy constants (LEC). In this project we will consider a Hamiltonian with a 3NF term that can be written as [6]

$$V_{3\text{NF}} = V_{3\text{NF},2\pi} + c_D V_{3\text{NF},1\pi\text{-ct}} + c_E V_{3\text{NF},\text{ct}}, \quad (1.3)$$

where c_D and c_E are two such LECs. The subscripts 2π , $1\pi\text{-ct}$ and ct refer to two-pion exchange, one-pion exchange and contact, and lastly three-nucleon contact interactions. The values of c_D and c_E have to be inferred from data and therefore have some degree of uncertainty, see for example [6] and [7]. How this uncertainty propagates to the computed ground-state energies of many-body systems is explored in section 8.3, where a study of the energy difference between the ground states of ^8Be and ^4He is of extra importance.

Chapter 2

The no core shell model

The many-body Schrödinger equation cannot be solved analytically, except in some special cases. Some form of approximation method is needed. The NCSM is one such method that is used in nuclear physics [1]. It is closely related to configuration-interaction methods used for many-electron systems [8].

2.1 The no core shell model basis

In order to represent equation (1.1) in a discrete model space, we need to define a truncated basis and project the many-body Schrödinger equation onto it. In the NCSM, the basis is constructed from harmonic oscillator single-particle states. The nucleons are fermions, so we must use an antisymmetric wave function to describe them. This is accomplished by antisymmetrizing the product of harmonic oscillator single-particle states using a Slater determinant.

The individual basis vectors in the many-body basis are denoted $|\Phi_i\rangle$ and the complete set is denoted $\{|\Phi_i\rangle\}_{i=1}^{\infty}$ [9]. This infinite basis captures all the information in the Schrödinger equation. But we cannot store an infinite number of states on a computer, so we must truncate it at some dimension $D(N_{\max})$. The cut-off parameter N_{\max} represents a total harmonic oscillator energy truncation. This user-supplied variable specifies the maximum number of single particle excitations above the lowest possible configuration. It is defined as [1, 2]

$$\sum_{i=1}^A (2n_i + \ell_i) - N_{\min} \leq N_{\max}, \quad (2.1)$$

where n_i (ℓ_i) is the principal (orbital angular momentum) quantum number of the i th nucleon, out of a total of A nucleons. The variable N_{\min} is given by

$$N_{\min} = \sum_{i=1}^A (2\tilde{n}_i + \tilde{\ell}_i), \quad (2.2)$$

where \tilde{n}_i ($\tilde{\ell}_i$) denotes the lowest possible configuration for the principal (orbital angular momentum) quantum number.

A visual representation of how the single particle harmonic oscillator basis is populated is shown in figure 2.1a. More specifically, it shows the lowest energy configuration in the harmonic oscillator basis for a nuclear system of four protons and four neutrons. Here we use the notation $N = 2n + \ell$. The $N = 0$ level is the lowest energy level, and larger values of N correspond to

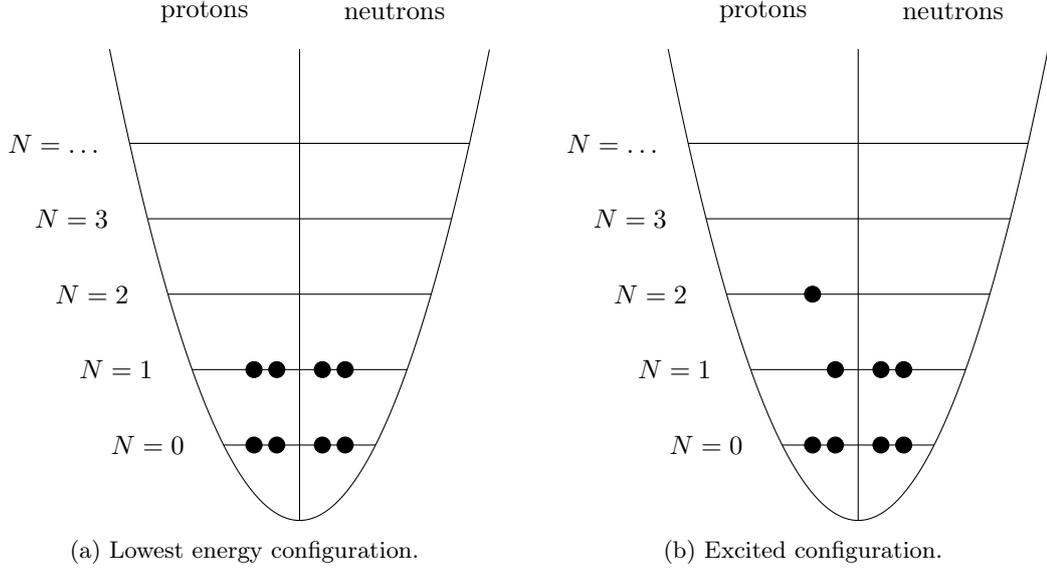


Figure 2.1: Visualization of a single particle harmonic oscillator basis for a system with four protons and four neutrons. The energy levels are indexed by the shorthand notation $N = 2n + \ell$.

higher energies. The basis is constructed from a three-dimensional harmonic oscillator (3D, HO), which has a degeneracy of

$$d_{3\text{D,HO}} = \frac{(N+1)(N+2)}{2} \quad (2.3)$$

for each level. Additionally, the particles are fermions with spin $1/2$, which means that we have two possible spin orientations (up and down) for each set of quantum numbers n and ℓ . This gives us a total single-particle, harmonic-oscillator (sp,HO) degeneracy of

$$d_{\text{sp,HO}} = 2 \cdot d_{3\text{D,HO}} = (N+1)(N+2). \quad (2.4)$$

Note that this degeneracy is per particle type, and we have two types, protons and neutrons. For example, the lowest energy level has $N = 0$, which gives a total degeneracy of $(0+1)(0+2) = 2$. Similarly, the $N = 1$ level has a degeneracy of 6. In this notation using N , equation (2.2) is written as $N_{\min} = \sum_{i=1}^A N_i$.

For the ${}^8\text{Be}$ case illustrated in figure 2.1a we have four protons and four neutrons. The lowest energy level has space for two particles of each species. The remaining four nucleons must be placed in a level with higher energy. If these extra nucleons are placed in the $N = 1$ level then the system is in the lowest possible energy configuration, which gives us

$$N_{\min} = \sum_{i=1}^8 N_i = \underbrace{0+0+1+1}_{\text{protons}} + \underbrace{0+0+1+1}_{\text{neutrons}} = 4. \quad (2.5)$$

Note that this is the lowest configuration in the single particle harmonic oscillator basis. It is not the same as the ground state of the many-body Schrödinger equation.

One or more of the particles may be excited, see figure 2.1b. Here, one of the particles in the $N = 1$ level has been excited to $N = 2$. This represents a higher energy than the configuration in figure 2.1a. It is also possible to have other excitations, up to and including

$N_{\text{totmax}} = \sum_{i=1}^A \hat{N}_i$, with \hat{N}_i denoting the highest allowed configuration (note the difference between N_{max} and N_{totmax}). This is how the total energy truncation parameter N_{max} limits the size of the basis. A large value of N_{max} is desirable for increased accuracy, but it also corresponds to a large basis set, with a correspondingly high computational cost. The notation $D(N_{\text{max}})$ represents the number of basis states of the total-energy truncated model space $\{|\Phi_i\rangle\}_{i=1}^{D(N_{\text{max}})}$.

The exact state vector $|\Psi\rangle$ for the system can be approximated by the NCSM state $|\Psi_k^{\text{NCSM}}\rangle$ [2, 10]

$$|\Psi\rangle \approx |\Psi_k^{\text{NCSM}}\rangle = \sum_i^{D(N_{\text{max}})} c_i |\Phi_i\rangle, \quad (2.6)$$

where c_i represents the amplitudes of the different basis configurations. If we use this NCSM state in the many-body Schrödinger equation (1.1) we get

$$\hat{H} \sum_i^{D(N_{\text{max}})} c_i |\Phi_i\rangle = E_k \sum_i^{D(N_{\text{max}})} c_i |\Phi_i\rangle. \quad (2.7)$$

We can then multiply from the left with $\langle\Phi_j|$, giving us

$$\sum_i^{D(N_{\text{max}})} \langle\Phi_j|\hat{H}|\Phi_i\rangle c_i = E_k c_j. \quad (2.8)$$

Once the quantities have been expressed in this basis, we can see that equation (2.8) is in fact a typical linear algebra matrix eigenvalue problem of the form $Av = \lambda v$, where A is a matrix, v is an eigenvector and λ is the corresponding eigenvalue. In the NCSM case we have matrix elements $\langle\Phi_j|\hat{H}|\Phi_i\rangle$ and vector elements c_i , as well as the energy eigenvalues E_k . These quantities then take the form of numerical values: a $D(N_{\text{max}}) \times D(N_{\text{max}})$ matrix for the Hamiltonian \hat{H} , a $D(N_{\text{max}})$ -dimensional vector for the state $|\Psi\rangle$ and a scalar for the corresponding energy eigenvalue E_k . The Hamiltonian matrix has a total of $D(N_{\text{max}})$ eigenvalues, so we would in fact need $D(N_{\text{max}})$ eigenvectors of dimension $D(N_{\text{max}})$, and $D(N_{\text{max}})$ scalar eigenvalues. Now that we have expressed our problem as a typical matrix-eigenvalue problem, we could in principle use known linear algebra methods for solving this system. However, using typical linear algebra methods is in this case impractical due to the rapidly increasing problem size, see section 3.1.

2.2 The Lanczos algorithm

As will be outlined in section 3.1, the method of direct diagonalization becomes intractable since the dimension $D(N_{\text{max}})$ of the matrix increases rapidly as the number of particles, or the parameter N_{max} , increases. Instead, iterative solution methods must be used. One such method, that is able to quickly approximate extreme eigenvalues in the spectra, is the Lanczos method [11]. This algorithm is well suited for working with very large matrices. In this project we are only interested in nuclear ground-state energies, so we only need the most extreme negative eigenvalue.

An outline of the Lanczos algorithm is presented in algorithm 1. The goal of this algorithm is to find the extreme eigenvalues of the H matrix. It does this by projecting the H matrix onto a smaller subspace spanned by so called Krylov vectors. This means that the essential characteristics of the large H matrix are captured in a smaller tridiagonal matrix T and a set of Krylov vectors $|K_k\rangle$. This smaller system T can then be diagonalized exactly to get approximations to the eigenvalues of the large matrix.

Algorithm 1: Outline of the Lanczos algorithm. Quoted algorithm and notation from [2].

Data: $D \times D$ symmetric matrix H , initial Krylov vector $|K_1\rangle$
Data: $|K_0\rangle = 0$, $B_0 = 0$
Result: Tridiagonal matrix T_k with diagonal elements A_i and off-diagonal elements B_j where $i = 1, \dots, k$ and $j = 1, \dots, k - 1$
Data: Iteration index k , starts at 1

```

1 while eigen-spectrum of  $T_k$  is not converged do
2    $|w_k\rangle \leftarrow \hat{H} |K_k\rangle$ 
3    $A_k \leftarrow \langle w_k | K_k \rangle$ 
4    $|v_k\rangle \leftarrow |w_k\rangle - A_k |K_k\rangle - B_{k-1} |K_{k-1}\rangle$ 
5    $B_k \leftarrow \sqrt{\langle v_k | v_k \rangle}$ 
6    $|K_{k+1}\rangle \leftarrow \frac{|v_k\rangle}{B_k}$ 
7    $k \leftarrow k + 1$ 
8   Reorthogonalize Krylov vectors if needed.
9 end
```

From a computational perspective, the Lanczos algorithm is a rather good candidate for parallelization in high performance computing (HPC) environments. Not all calculations in the algorithm are inherently parallel, but the most time-consuming one, the matrix-vector product on row 2, is. Calculating this product is a well understood process that has favorable properties for parallelization. There are many software libraries for computing smaller matrix-vector products in parallel. The problem in this project is that the matrix is so large that it cannot be handled directly by this type of libraries, a special solution for this particular use-case is needed. But the underlying parallelizable property of the matrix-vector product still remains.

2.3 Computation of large matrix-vector products in the Lanczos algorithm

One important property of the Lanczos algorithm is that it does not need to explicitly store the entire matrix. It just needs to be able to compute the product of the matrix with some vector. The matrix elements in our specific Hamiltonian matrix can be generated on the fly from 2NFs and 3NFs, together with transitions densities. The elements are generated when they are needed in the computations. This means that the matrix-vector product can be computed without storing the full Hamiltonian matrix, as long as we have access to the data needed to generate the matrix elements. This is sometimes referred to as an implicit matrix. In JupiterNCSM (see

section 2.4), the matrix-vector product is implemented according to [2]

$$\begin{aligned}
 |w_k\rangle &= \hat{H} |K_k\rangle \\
 &= \sum_{i=1}^{D_\pi} \sum_{l=1}^{D_\pi} \sum_{j=1}^{D_\nu} \sum_x K_{n(i,j)} (M_x^{\text{int-}pp} + M_x^{pp}) \langle \pi_l | \hat{t}_x^{pp} | \pi_i \rangle (|\pi_l\rangle \otimes |\nu_j\rangle) \\
 &+ \sum_{i=1}^{D_\pi} \sum_{j=1}^{D_\nu} \sum_{m=1}^{D_\nu} \sum_x K_{n(i,j)} (M_x^{\text{int-}nn} + M_x^{nn}) \langle \nu_m | \hat{t}_x^{nn} | \nu_j \rangle (|\pi_i\rangle \otimes |\nu_m\rangle) \\
 &+ \sum_{i=1}^{D_\pi} \sum_{l=1}^{D_\pi} \sum_{j=1}^{D_\nu} \sum_x K_{n(i,j)} M_x^{ppp} \langle \pi_l | \hat{t}_x^{ppp} | \pi_i \rangle (|\pi_l\rangle \otimes |\nu_j\rangle) \\
 &+ \sum_{i=1}^{D_\pi} \sum_{j=1}^{D_\nu} \sum_{m=1}^{D_\nu} \sum_x K_{n(i,j)} M_x^{nnn} \langle \nu_m | \hat{t}_x^{nnn} | \nu_j \rangle (|\pi_i\rangle \otimes |\nu_m\rangle) \\
 &+ \sum_{i=1}^{D_\pi} \sum_{j=1}^{D_\nu} \sum_{l=1}^{D_\pi} \sum_{m=1}^{D_\nu} \sum_{x,y} K_{n(i,j)} M_{x,y}^{pn} \langle \pi_l | \hat{t}_x^p | \pi_i \rangle \langle \nu_m | \hat{t}_y^n | \nu_j \rangle (|\pi_l\rangle \otimes |\nu_m\rangle) \\
 &+ \sum_{i=1}^{D_\pi} \sum_{j=1}^{D_\nu} \sum_{l=1}^{D_\pi} \sum_{m=1}^{D_\nu} \sum_{x,y} K_{n(i,j)} M_{x,y}^{ppn} \langle \pi_l | \hat{t}_x^{pp} | \pi_i \rangle \langle \nu_m | \hat{t}_y^n | \nu_j \rangle (|\pi_l\rangle \otimes |\nu_m\rangle) \\
 &+ \sum_{i=1}^{D_\pi} \sum_{j=1}^{D_\nu} \sum_{l=1}^{D_\pi} \sum_{m=1}^{D_\nu} \sum_{x,y} K_{n(i,j)} M_{x,y}^{pnn} \langle \pi_l | \hat{t}_x^p | \pi_i \rangle \langle \nu_m | \hat{t}_y^{nn} | \nu_j \rangle (|\pi_l\rangle \otimes |\nu_m\rangle).
 \end{aligned} \tag{2.9}$$

This equation looks quite complex, but it is constructed from only a few basic parts. On the first line we have a compact representation of the matrix-vector product in the Lanczos algorithm. A Hamiltonian operator \hat{H} (a matrix) operates on the state $|K_k\rangle$ (here a Krylov vector) to produce a new state $|w_k\rangle$ (a new vector). The second line and below shows this operation in more detail. The variable $K_{n(i,j)}$ represents vector elements in our Krylov vector, the M variables represent matrix elements and the brackets $\langle |\hat{t}| \rangle$ with the \hat{t} operator are transition densities. The kets at the end with $|\pi\rangle$ and $|\nu\rangle$ are the proton and neutron Hilbert bases. They are multiplied together to form the full Hilbert space for the system. The variables D_π and D_ν represent the dimensions of the proton and neutron bases. We also have the x and y variables, which are abstractions for sets of single-particle quantum numbers. The p and n superscripts indicate if the terms operate on the proton or neutron spaces, respectively. The int superscript indicates that the term is related to the internal kinetic energy.

As we can see in this equation, the matrix-vector product is just a large number of combinations of a smaller set of precomputed input data. This is what it means that the matrix is implicitly constructed. For full details about how this equation is derived, see chapter 4 of [2].

From this equation we can also see that the required set of precomputed data consists of matrix elements and transition densities. The Krylov vectors K are part of the Lanczos algorithm and are constructed as part of algorithm. The proton and neutron bases are part of the proton-neutron formalism, where protons and neutrons are considered to be separate species of fermions [12]. It is also possible to treat protons and neutrons as the same type of fermion, with different values of the isospin quantum number. The advantage of using proton-neutron formalism is that the two individual bases are much smaller than the corresponding basis in isospin formalism (for a typical nucleus with a combination of both protons and neutrons).

Additionally, we can see that the matrix-vector product naturally divides itself into many partial sums of products between Krylov vectors, matrix elements and transition densities. These

sums then become very natural work items when parallelizing the matrix-vector product. If each core of the central processing unit (CPU) takes responsibility for a few of these items, then the results of all of the operations can be summed up in the end to get the final result.

2.4 JupiterNCSM

Much of the work in this project is related to the NCSM solver JupiterNCSM [2]. It has previously been used to accurately compute ground-state energies for nuclei with $A \leq 6$ nucleons [6]. The program covers some of the main parts of the NCSM solution method, but the code also requires input data from other research codes. For example, the 2NF and 3NF matrix elements in a harmonic oscillator basis are required as a given input. JupiterNCSM also uses a many-body basis computed by pAntoine, which is then transformed to a state usable by JupiterNCSM by the Anicre code [2]. When provided with these inputs, JupiterNCSM computes energy eigenvalues and eigenvectors to the nuclear Hamiltonian for the given nucleus.

The JupiterNCSM code is structured around several somewhat independent subprograms. The programs Neptune and Mercury transform the precomputed 2NF and 3NF data from so called JT- and J-scheme to M-scheme, which JupiterNCSM uses internally. For more details, see section 4.1.4 in [2].

One of the main subprograms of interest in this project is Bacchus. This program runs the Lanczos algorithm based on input data prepared by other subprograms in the JupiterNCSM toolchain. Bacchus is closely coupled to the subprogram Minerva, which specializes on computing large matrix-vector products. Most of the work in the current project is performed in Bacchus, but the other programs are essential for the correct functioning of the complete toolchain.

There are also two smaller programs, Mars and Vulcan, that are of interest in this project. Mars is an early attempt at improving the matrix-vector product work order. In this project we create a new program, Venus, that improves upon and replaces the Mars software, for details see section 5.1. Vulcan is used to perform addition of separate terms in the nuclear Hamiltonian, see section 1.1. This addition of terms is an important part of the statistical analysis in section 8.3, but the program itself is trivial and is therefore not discussed further.

There are a few additional subprograms that are part of the JupiterNCSM system, but they are only used to a small degree in this project. A more complete description of the programs is available in [2].

Chapter 3

Parallel computing and computer memory

Chapter 2 outlined how the NCSM method can be applied for solving the Schrödinger equation for the nuclear many-body problem. The chapter focused mainly on the physics and mathematical equations that are needed to solve the problem. However, using the NCSM to describe the nucleus is a very computationally demanding task, so we also need to consider the computational aspects. This chapter presents the main computational difficulties in this project and how they are solved, or at least how they are made manageable for current computer systems.

3.1 Outline of computational difficulties

One of the main problems when using the NCSM method is that the memory required to store the Hamiltonian matrix increases rapidly as the dimension increases. One of the specific aims of this project is to simulate ${}^8\text{Be}$ at $N_{\text{max}} = 8$, which means that we must diagonalize a matrix with $D \approx 10^7$. The entire matrix would then consist of $D^2 \approx 10^{14}$ matrix elements. This is too many elements to be handled directly by common linear algebra software libraries. Fortunately, the matrix is relatively sparse, having in the order of $D^{3/2}$ non-zero matrix elements for the case including 3NFs [13]. Thus, we have approximately $D^{3/2} \approx 1.1 \times 10^{11}$ non-zero elements. Storing this as double-precision floating point numbers, with a size of 8 bytes each, would require approximately 9×10^{11} bytes = 900 GB of memory for the matrix elements (plus additional memory for saving the matrix indices). A more reasonable number, but still too large to be saved explicitly.

Another useful property is that the matrix elements do not need to be explicitly stored. The non-zero elements can be generated on the fly from a smaller set of precomputed data. This favorable property makes it possible to divide the matrix into smaller blocks and perform computations with those instead. The computer then only needs to store one block of precomputed matrix data and a small section of a vector at any given time. The computations are then completed in blocks and in the end collected into the final result. These properties reduce the intractable problem into one that is manageable with currently available computer systems [14]. Using these properties, the JupiterNCSM code is able to reduce the storage size of this matrix to around 200 GB.

3.2 Description of a typical high performance computing cluster

Typical desktop computing is the type of computing most people are familiar with. It generally involves a single desktop computer or laptop used by a single person at a time. This usage is enough for many tasks, but not for the heavy calculations required in this project. Instead, it is beneficial to use one of the many HPC clusters available to researchers. Using these, the user can access much better computational performance than what is available on a single workstation. These HPC clusters are designed to run programs that use a large amount of resources in terms of time, CPU power, memory or similar.

HPC systems are generally composed of many regular computers (nodes) interconnected to each other via high-throughput, low-latency network interconnects. These individual nodes can, in terms of hardware, be similar to powerful desktop workstations or servers. However, the variations in hardware configuration can be quite large since there is no standard configuration. The HPC system Tetralith, that is used in this project, is composed of 1908 compute nodes, where each node has two Intel Xeon Gold 6130 server CPUs, see table 7.1 [15]. Access to Tetralith is shared among researchers of several universities, so a single user is typically only able to use a smaller part of the system at any given time.

HPC systems also have other special properties, such as the ability to run user-supplied programs in parallel on several nodes at the same time. If these programs are structured correctly, then the nodes can cooperate to solve large problems in less time than a single computer would be able to. This ability is not unique to HPC systems, but it is a very prominent characteristic of them.

3.3 Memory in high performance computing systems

Most current computer systems have several layers of memory, each with different performance characteristics. Consider first the CPU of a single computer node. It performs all the calculations that we need in our simulations. Inside, it has a set of very small memory caches, with storage sizes typically in the kB to MB range, which are used as very fast, short-term storage for data that is currently in use. The CPU is connected to a larger primary memory, often referred to as random access memory, RAM. This type of memory is typically not as fast as the CPU caches, but it is much larger, in the order of tens to a few hundred GB. This makes it suitable for storing active application data. However, as we could see from section 3.1, RAM is generally not large enough to store all the data required for diagonalizing the Hamiltonian matrix.

Computers typically also have secondary memory in the form of a hard disk drive, HDD, or a solid state drive, SSD. Storage on these devices is not as fast as on RAM, but is generally much larger, typically in the range of a few hundred GB to a few TB. SSDs are faster than HDDs, but are usually more expensive. The average throughput rate achieved while simulating ^8Be at $N_{\text{max}} = 8$ with JupiterNCSM on a thin, large-disk node on the Tetralith cluster, see table 7.1, is approximately 0.44 GiB/s. This type of node has a non-volatile memory express (NVMe) scratch disk.

If node-local storage on secondary memory is not enough, then it is also possible to use an array of network attached disks to emulate an even larger file system. This allows the use of huge amounts of storage space, the total size of which is mainly limited by financial considerations. For example, the storage system used by Tetralith has a size of a few PB [16], but since it is a shared resource a single project is not able to use all of it. An allocation in the order of up to a few hundred TB are reasonable for larger research projects in Sweden at the moment [17]. The

throughput of this type of memory depends to a large degree on the file access pattern. If a single, large file is processed then it might have performance comparable to, or even exceeding, the node-local HDDs or SSDs. However, if a large amount of small files are processed, then performance could be significantly lower, especially compared to node-local SSDs. When simulating the ^8Be , $N_{\text{max}} = 8$ case with JupiterNCSM we get an average throughput rate of approximately 0.125 GiB/s when using the network attached disks on one of the storage systems at Tetralith.

From this summary it is clear that there is a trade-off between storage space and data throughput. Fast memories such as CPU caches are typically small, while large memories such as network attached disks can be slow. In the current project we are interested in getting the best possible computational performance. The best performance translates, in this case, to the highest possible data throughput rate. Our data is large, so it will not fit in the fastest (but smallest) memories. Slower, but larger, storage must then be used to some degree in order to perform the calculations. It is possible to use a combination of storage systems, which is what is typically done for problems with large amounts of data. In this project we store the input data on secondary storage and then load it into primary storage for processing as needed. This combines the large storage space of the secondary storage with the fast data throughput rate of the primary storage.

3.4 Methods of parallelization

The simplest form of computation is the single-threaded processing. In this case, all computations are done by a single CPU core. This method is conceptually simple to understand and implement, but it is generally not the most efficient way of performing calculations. Instead, most modern computer systems utilize parallelization to some degree, often using the multi-core approach. The CPU in this type of system has two or more CPU cores that are able to concurrently perform several independent computations. Ideally, this could increase performance linearly with the number of independent cores. However, this approach introduces a stronger need for synchronization, work division and sharing of resources that usually limits the overall performance increase. How prevalent this punishment is depends on several factors, such as the type of computer running the calculations, as well as the kind of tasks being performed. Performance is typically better if work items are completely independent of each other and fit in core-local cache. The performance penalty can, however, be quite severe if the calculations depend on a slow external resource, such as files saved on an HDD.

Developing multi-threaded applications is generally more cumbersome than writing single-threaded ones. A common first step when parallelizing a program is to adapt it so that it can make use of the multi-core architecture available on most modern CPUs. An application programming interface (API) that can make this process simpler for the programmer is OpenMP (Open Multi-Processing) [18]. Using this API, the programmer defines which sections of code should be parallelized, and the API takes care of how this is achieved in practice. OpenMP performs parallelization on a single computer node, meaning that it works in a shared-memory architecture.

Even further parallelization can be achieved by using several computers at once. Using parallelization between separate nodes requires a different approach than between cores in a single CPU. Parallelization between nodes is a distributed-memory system, as opposed to the shared-memory system used by OpenMP, and therefore implies the need for inter-node communication. A commonly used communications standard for facilitating program execution across several network-attached computers is the Message Passing Interface, MPI [19]. The goal of using MPI is to get more parallelization than what is possible on a single computer node. If we use only a

single computer node, then performance is essentially limited by the speed of a single component in a single computer. If we use many computers, the parallel performance is limited by the number of nodes that we manage to recruit, and also how well the workload can be parallelized.

3.5 OpenMP

OpenMP is an API that simplifies the development of certain types of multi-threaded programs. It is often used in scientific computing to parallelize execution of loop iterations that are independent of each other. It is possible to write parallel code without help from specialized APIs, but it typically requires significantly more developer effort. The developer has to manually keep track of which thread is doing what and make sure that the different threads are not interfering with each other. One of the aims of OpenMP is to hide some of these details from the developer. The task of the developer is then to indicate to OpenMP which parts that can efficiently be parallelized. The developer also has to know how to separate the threads from each other, so they do not interfere with each other. Other than that, OpenMP takes care of the low-level details. This frees up developer time that can be spent on other tasks or improving other parts of the program. Introducing OpenMP parallelization is a common way of speeding up single-threaded programs without having to spend too much time rewriting existing code.

3.6 The Message Passing Interface, MPI

The MPI is a specification for how nodes in a computer network should communicate and cooperate to execute a given program. The nodes can be interconnected using regular Ethernet connections, or using specialized communication systems such as Intel Omni-Path [20] or Infini-band [21]. There are several implementations available of the MPI standard, for example MPICH [22] and Open MPI [23], but since the interface is standardized, there should not be any issues when switching between compliant implementations of the standard.

In order to allow efficient collaboration, programs wishing to use more than one computer node need to be structured in a specific way. The MPI specification provides rough guidelines for how this can be achieved. It is then up to the developer to adapt any programs to this general structure.

With the general structure in place, the developer can use an implementation of MPI to take advantage of the communication algorithms that are specified in the MPI standard. These algorithms range from simple messages between two nodes to collective communication between all nodes at the same time.

It is of course possible to implement this functionality without MPI. However, one of the main benefits of MPI is that it reduces the workload of the developer. It hides many of the complexities of initializing and maintaining communication in a computer network. This saves development time, which can be directed at other tasks. MPI is also an industry standard technology, which means that it has been thoroughly tested and evaluated. An implementation of MPI is generally available at most HPC centers.

3.7 Hybrid parallelization: OpenMP and MPI

By combining the OpenMP and MPI technologies, it is possible to leverage both types of parallelization. On a lower level, we utilize the multi-core nature of modern CPUs using OpenMP, as well as the more higher-level parallelization that comes from connecting several such computers

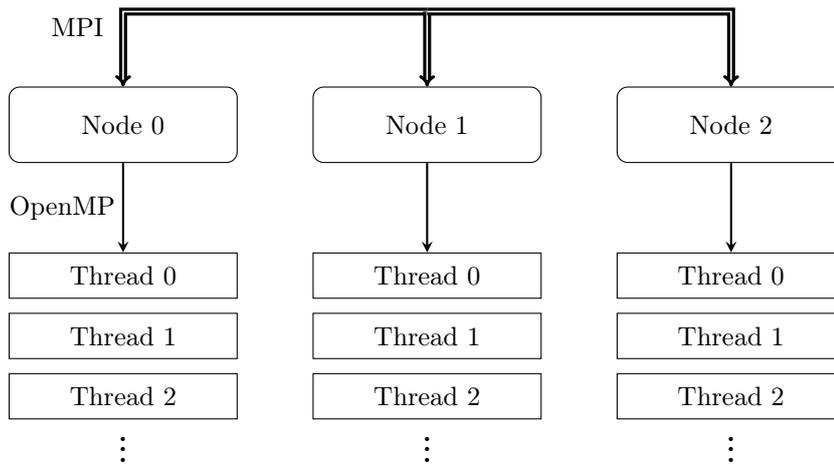


Figure 3.1: Schematic representation of hybrid parallelization via the combination of (distributed memory) MPI and (shared memory) OpenMP.

to a network, with communication using MPI. This combination of OpenMP and MPI is quite common in scientific computing and is known as hybrid parallelization.

A schematic representation of how this parallelization is structured is presented in figure 3.1. It shows, in this case, three nodes connected to each other using MPI. Within each node we have parallelization between threads using OpenMP. Together they give us two levels of parallelization.

3.8 Hybrid parallelization in JupiterNCSM

The original version of JupiterNCSM has OpenMP parallelization for some of the more computationally heavy tasks. For example, the matrix-vector product is calculated in blocks using OpenMP parallelization.

One of the main improvements in this project is the extension of JupiterNCSM with MPI capabilities for running the Lanczos algorithm. The blocks in the matrix-vector product can then be distributed between nodes on a network. One node, the server node, is responsible for coordinating tasks, distributing input data and collecting partial results from the other nodes. The other nodes calculate partial sums (work items) using data from the server node. When they are done, they return their partial results to the server node and wait for new input. All of this communication is achieved using algorithms specified in the MPI standard.

The addition of MPI capabilities to JupiterNCSM means that this research code now can leverage the advantages of the OpenMP and MPI combination.

Part II

Implementation

Chapter 4

Profiling

The efficient use of available computer resources is an important area to explore for future use of the tool. Large-scale JupiterNCSM runs can take days or more to run. It is therefore crucial that the code uses computer resources as efficiently as possible. Investigating code performance, especially when looking for bottlenecks, is often known as code profiling. In this chapter we explore the main profiling and optimization efforts performed in this project.

4.1 Profiling single instructions

One of the main objectives in this project was determining the main performance bottleneck in JupiterNCSM. One of the first tasks was then to evaluate which portions of the code are the slowest. The conventional wisdom when optimizing code for performance is to start by profiling the code. This gives objective measurements of which parts of the program take the longest time to run. These parts are then good candidates for performance optimization. By using this methodology, the potential performance gains are maximized while at the same time limiting the required development effort.

The profiling efforts were focused on the Lanczos algorithm implementation Bacchus. It is a code that can take quite a significant amount of time to run. Good run-time performance is then an important factor. This solver uses the code Minerva to calculate the matrix-vector product using the smaller blocks, or partial sums, defined in equation (2.9). Each such block can be seen as an individual instruction. The execution of each of these instructions has a few different stages. These stages define the general characteristics of the workload throughout the computation. At the start, a thread is assigned a certain instruction to compute. This thread is then placed in a queue to allocate primary memory for the required data files (vector files, index lists and matrix data). If JupiterNCSM has RAM available for these files, then it immediately allows the thread to allocate the required memory. If not, then old files are evicted from RAM to make room for new ones. It is important that this process does not evict files that are currently in use. This time for memory allocation is one of the investigated parameters.

Then comes a phase of reading files from disk. Since we are working in a multi-threaded environment, then we cannot guarantee in advance which files have already been loaded into primary memory and which ones have not. If the files are already in RAM, for example if they were used in the computation of a previous block or if another thread is currently using them, then they are directly re-used, without disk access. If another thread is currently working on loading them into primary memory, then this thread just waits for that process to finish. On the other hand, if the required files are not in main memory, and they are not currently being

read by another thread, then this thread starts reading them from disk. Both the time waiting for other threads to finish reading files, and the time spent actively reading files is measured.

Finally, time spent performing useful calculations on the CPU is also recorded. This is the time used for actually performing the multiplications in the matrix-vector product.

The implementation of instruction-level profiling of the four main run-time parameters described above is one of the contributions of this project. This profiling data is stored to a file on disk, and can be used to study program execution in greater detail than what has previously been possible. Due to the nature of how parallel computer systems work, timing information will overlap between instructions. This corresponds to how parallel cores execute instructions simultaneously. From this information it is possible to construct a timeline over a complete JupiterNCSM run. This timeline can be used to both diagnose possible performance problems and study effects of proposed optimizations. We will in section 7.3 use this profiling data to understand the main performance bottlenecks in JupiterNCSM.

4.2 Study of number of files and file sizes

One aspect of the code that has impact on usability is the number of files required. One of the programs in the JupiterNCSM toolchain, Anicre, does produce an excessive number of very small files. Figure 4.1 shows the number of needed data files as a function of N_{\max} for three different nuclear systems. It is clear that even the small ${}^8\text{Be}$, $N_{\max} = 2$ case generates several thousand files. The $N_{\max} = 8$ and $N_{\max} = 10$ cases consist of well over a hundred thousand files.

This large number of files can be cumbersome to handle, especially if the storage medium has limited IOPS (input/output operations per second) performance. For example, typical HDDs might be able to perform in the order of 100 IOPS [24]. A large fraction of the simulation run time would then be dedicated to just opening very small files. This problem is somewhat mitigated by using SSDs, which can achieve rates in the order of 100 000 IOPS or more [25]. Even so, a large number of files is still undesirable and can cause problems with, for example, storage allocations on HPC systems. It would definitely improve usability if the number of files could be reduced. However, other improvements were considered of higher importance and the file number issue was left for future work.

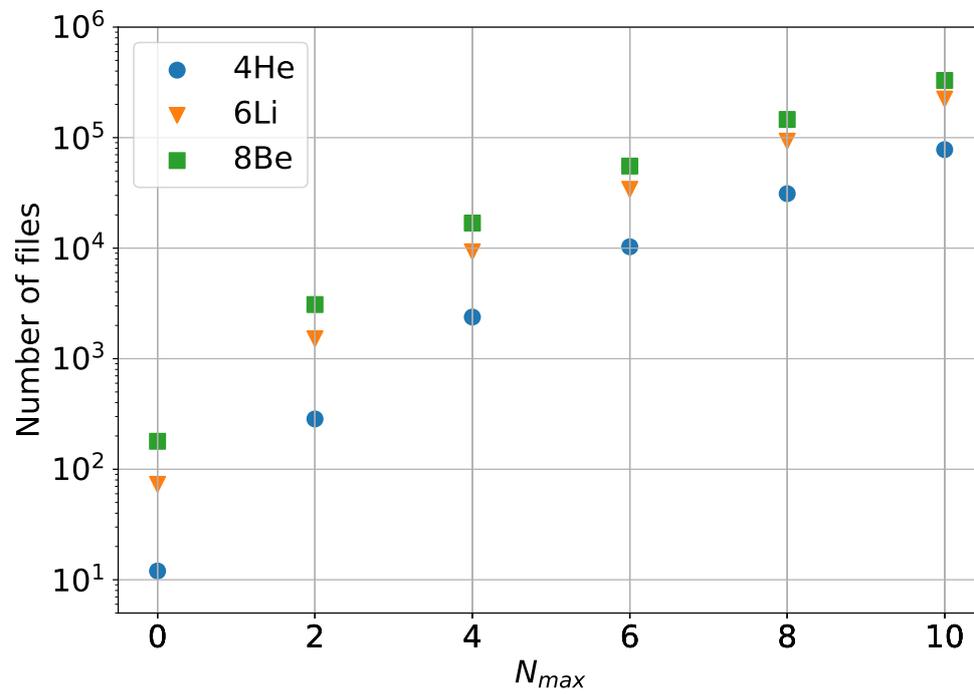


Figure 4.1: Number of implicit matrix element data files as a function of N_{max} for ^4He , ^6Li and ^8Be .

Chapter 5

Work order for the matrix-vector product

In this chapter we explore improvements made to JupiterNCSM that relate to the disk data bandwidth bottleneck. In section 5.1 we present an improved work order (also called evaluation order) that minimizes the amount of data that must be re-read from disk in each Lanczos iteration. Section 5.2 discusses a new file division scheme that allows JupiterNCSM to use matrix data spread out over several node-local scratch disks to avoid using slow network attached disks. Finally, section 5.3 introduces a disk load estimation program that can be used to estimate simulation run times.

5.1 Improved work order

One of the most important performance factors in JupiterNCSM is the locality of data. Performance is good if data files can be kept in RAM, but it suffers when files must be loaded from disk, which is the case for the larger simulations in this project. It is therefore desirable to minimize the amount of data that JupiterNCSM must load from disk. Certain files are re-used in several partial sums of the matrix-vector product. It would then be beneficial if these files could be loaded, fully processed, and lastly evicted from RAM to make room for new files. This way, the file is only loaded once, thereby minimizing the amount of data that must be re-read from disk.

For reference, in the ${}^8\text{Be}$, $N_{\text{max}} = 8$ case, JupiterNCSM reads data from disk equivalent to about three times the total size of the input data in each Lanczos iteration, see figure 7.6. This type of behavior is true for cases where the total size of the input data is significantly larger than the available RAM. The fact that JupiterNCSM reads the same data several times hints to possible inefficiencies in the way the software processes data.

The most direct way of overcoming this inefficiency is to reorganize the order in which the instructions are processed. The instructions are independent of each other, so the order in which they are evaluated does not affect the final result. The order in which they are processed is defined in the `evaluation_order` file. An early attempt at optimizing this order was performed in the Mars program in JupiterNCSM [2].

The problem of finding an optimal work order is a type of Traveling Salesman Problem (TSP). Finding an exact solution to this problem is conceptually easy but computationally very hard. The brute-force approach is to test every single possible order and then pick the best one. This method is only practical when the number of TSP graph nodes is very small. However, in this

project we often have millions of instructions (TSP nodes) that must be visited. Exact solutions are then not possible to find.

In this project we developed a new application-specific heuristic for approximating the optimal solution to this TSP. First, we sort all input data files by decreasing file size. Then, we focus on the largest file. We find all instructions that reference this file and place them first in the `evaluation_order` file. While processing these instructions we must also access other, smaller, data files. The large file is placed in RAM for as long as this set of instructions is being evaluated. The small extra files could be loaded for a single instruction, and might then not be needed for a while. They are then evicted from RAM and will eventually be reloaded again, when required. Reloading these small files a few times is cheaper than reloading the large file. When the large file has been processed we move on to the second-largest file and complete the corresponding instructions. We continue this process until we have processed all instructions. This way, we minimize the number of loads that have to be performed for large files. This heuristic gives us an evaluation order that is a much better approximation of the optimal solution to the TSP than what was previously available in JupiterNCSM using the Mars program. Since this way of computing the evaluation order differs a bit from the old version in Mars, it is also fitting that the program receives a new name: Venus. A comparison of the performance of the two evaluation orders computed using Venus and Mars is presented in section 7.4.

5.2 File division between nodes using Venus

As was described in section 3.3, the type of storage used can have a significant impact on the disk read data throughput rate. For a single node on the Tetralith HPC system, a node-local NVMe scratch can deliver around three times the average disk read throughput rate compared to an array of network attached disks. It is then very favorable to keep data on node-local scratch disks. Unfortunately, these disks are not large enough for the input data required by the ^8Be , $N_{\max} = 10$ case, see figure 7.1, which we want to be able to run in the future. We are then left with two options. The first one is to use network attached storage and accept a penalty to performance. The second one is to find a way to divide the files between several MPI nodes. That way, we can use the combined NVMe scratch-disk space of several nodes to store the data.

One of the interesting properties of the evaluation order heuristic described in section 5.1 is that the larger files are processed in a concentrated manner. For example, the largest file is only processed at the very beginning, and then not used again in this iteration. This means that we could, in principle, make sure that a single MPI node has this file on its scratch disk. None of the other nodes would need to have this file. Similarly, the second-largest file could then be assigned to the second MPI node. We could continue this process until all files have been assigned to a particular MPI node. Some files would inevitably be duplicated across MPI nodes, but no single node would have to store the entire input data. We would then, with enough MPI nodes, be able to store input data that is larger than the size of individual scratch disks.

Venus saves the names of the needed index list and matrix element data files for each MPI node as plain text files. File names required by the node with MPI world rank 0 are saved in the files `index_list_indices_0` and `matrix_element_indices_0`. Similarly, the node with MPI world rank 1 uses `index_list_indices_1` and `matrix_element_indices_1`, etc.

Using multiple MPI nodes also requires instructions to be divided across all MPI nodes. Venus does this by having separate instruction queues for each MPI node. These queues are then stacked on top of each other in the end of the division process to form a single `evaluation_order` file. This file is then structured in a similar way as in the single-node case, but now it is accompanied by the `evaluation_order_indices` file which describes which MPI node should handle which

instructions, see section 6.2.

5.3 Disk load estimation program

Since we know the size of all files, and we know the order of appearance in the `evaluation_order` file, we can estimate the disk load we are going to encounter when we run the simulation. Knowing this could be especially useful in the large N_{\max} cases, where simulations take a long time to complete. It is also useful for estimating if a certain simulation is feasible.

To calculate this approximation, we developed a small program that mimics the internal memory handling system in the Bacchus subprogram of JupiterNCSM. It pretends to load matrix data files, and then handles them in memory just as Bacchus would, but without doing any NCSM calculations. While doing this, it records the amount of data read from disk. The end result of this program is a simulation of the total amount of data read from disk for a single Lanczos iteration. This information together with average data read rates for JupiterNCSM simulations can be used to predict run times, at least under the assumption that JupiterNCSM continues to be bottlenecked by disk data throughput rate.

Chapter 6

MPI parallelization

The original version of the JupiterNCSM software had only OpenMP parallelization on a single computer node. This enabled some degree of computational parallelism, but computations were limited to the performance of a single node. An additional level of multi-node parallelization was identified as a requirement for the study of nuclei with a larger number of nucleons [2]. In this chapter we begin by describing how JupiterNCSM translates the partial sums in equation (2.9) to the instructions that are processed in parallel using the OpenMP parallelization. This is described in section 6.1. Then in section 6.2 we outline how this underlying structure is used to introduce a distributed-memory architecture using MPI. And finally, section 6.3 describes an MPI-compatible resume feature that was added to the software.

6.1 From partial sums to JupiterNCSM instructions

As described in section 2.3, JupiterNCSM performs the matrix-vector product in the Lanczos algorithm in smaller blocks and then combines the result in the end. The fact that this product can be performed in blocks is very advantageous because it allows us to divide the workload between independent computer nodes. The OpenMP parallelization utilizes this property to some degree, but only within a shared-memory architecture. In order to maximize the use of current computer systems, JupiterNCSM must be able to distribute work items in a distributed-memory architecture.

We start by describing how the mathematical operation in section 2.3 is translated into computer code. The large matrix-vector product involves two Krylov vectors, (input and output ones), one or two index lists and one matrix element file. Each of the variables in equation (2.9) is represented by a file on a storage device on the computer. Each file is also identified by an integer index. Every instruction (partial sum) in equation (2.9) can be broken down into a set of four or five file indices. The instructions indicate how to combine the data files to create one partial sum. This list of instructions is stored as four or five space-separated indices in rows in a plain text file, by default called `evaluation_order`. The set of all partial sums is then added together to form the complete solution. All information that we need to perform the complete matrix-vector product is therefore contained in the data files for each factor in equation (2.9), and in the list of instructions.

An excerpt from an `evaluation_order` file for ${}^8\text{Be}$, $N_{\max} = 0$ is shown in figure 6.1. Each row represents an individual instruction that must be processed. Notice that the rows have either four or five indices. The first two integers after the `BLOCK:` prefix represent the indices of the input and output Krylov vector blocks. If a row has only four indices, then the third index

identifies a particular index list (also called transition density). If a row has five indices, then the third and fourth integers represent index lists. The last index (fourth or fifth, depending on length) represents the required matrix element file.

```
BLOCK: 3 4 38 134 113
BLOCK: 4 5 46 141 113
BLOCK: 2 3 65 18 66
BLOCK: 3 4 87 39 66
BLOCK: 4 5 94 47 66
BLOCK: 2 2 158 159
BLOCK: 1 1 161 159
...
```

Figure 6.1: Excerpt from an `evaluation_order` file for ${}^8\text{Be}$, $N_{\text{max}} = 0$. The four or five integers correspond to file indices in the input data. The columns represent input Krylov vector, output Krylov vector, index list 1, index list 2 (not included in all instructions), and finally a matrix element file.

The instructions in the `evaluation_order` file could in principle be computed in order, one at a time, by a single thread. This would work, but it would give poor computational performance. Instead, the instructions are processed in parallel using OpenMP, where each instruction is assigned to a separate thread. The matrix-vector product is then computed in parallel, usually with the same number of threads as there are physical CPU cores on the computer. Ideally, this would mean that simulation run time is inversely proportional to the number of CPU cores used.

Since the instructions are independent of each other, it is then possible to parallelize the program even further by letting several computer nodes cooperate on computing the same matrix-vector product. For example, if two computers are used, then the first computer could process the first half of the instructions in the `evaluation_order` file, while the second computer processes the second half of the instructions. These halves are processed using OpenMP, just as before. If both computers have similar computational performance and the instructions take approximately the same time to complete, then they should be able to perform the matrix-vector product in about half the time. Unfortunately, not all instructions take the same amount of time to complete, which complicates the reasoning a bit. Some effects of this are studied and mitigated in section 5.1. These multi-node computations could be achieved by adding distributed-memory MPI parallelization on top of OpenMP. Note that the main performance bottleneck is not CPU performance, but rather the storage medium data bandwidth. The main performance gains then come from increasing parallelism in the secondary storage data read performance. Some gains also come from the added RAM space when using several nodes.

Since the results of each operation is added to the (node-local) output Krylov vector, this vector must periodically be synchronized across nodes. This requires a reduction operation followed by a broadcast operation. In other words, at the start of each parallel matrix-vector product computation, the server node broadcasts its version of the input Krylov vector to all the worker nodes. The worker nodes then use this input vector in their computations, and gather partial results in a node-local output Krylov vector. At the end of the matrix-vector product operation, the server node collects partial results (the node-local output Krylov vectors) from all the nodes and creates a unified version of this output Krylov vector using a reduction operation. This vector can then be broadcast as the input Krylov vector in the next Lanczos iteration, which restarts the cycle.

6.2 MPI implementation details

The MPI parallelization implemented in this project is structured in a way that minimizes the amount of changes that must be introduced to the JupiterNCSM code. The original version of the code was already structured in a way that allows for OpenMP parallelization, so an important foundation and thread-safe synchronization mechanism was already in place for a shared-memory architecture. What was needed was synchronization and sharing of data between nodes in the new distributed-memory architecture.

Figure 6.2 shows a graphical representation of the main steps in the MPI parallelization. The server node¹ is responsible for initializing the environment and communicating the initial Krylov vector to all other nodes. This communication is performed using the MPI broadcast operation. The nodes then read two files, the `evaluation_order` file and the `evaluation_order_indices` file. The first file indicates all instructions that must be performed and the second file specifies which instructions should be processed by which worker node. It essentially specifies a start and a stop index in the `evaluation_order` file that each node should process. The indices in the `evaluation_order_indices` file are non-overlapping, which ensures that each instruction is completed exactly once. When each node has completed the assigned instructions, it waits for the other nodes to finish, then they collectively reduce the node-local output Krylov vectors to a unified version on the server node. This communication is performed using the MPI reduce operation. The reduction operation is a sum over all node-local versions of the output Krylov vector. The server node then checks if the Lanczos convergence criteria has been met. If it has, then the process ends. If not, then it starts another iteration.

Internally, the division of instructions between nodes is handled using instruction indices and iterators. JupiterNCSM contains an iterator that points to the next instruction to be processed. This works well in the shared-memory OpenMP environment, but must be modified slightly in the distributed-memory MPI environment. On each node, JupiterNCSM looks at the local MPI world rank and then reads the block of instructions that it should process from the `evaluation_order_indices` file. It then sets the internal instruction iterator to point to this first instruction. After that, it sets a variable for the maximum number of elements so that the iterator does not go further than the last assigned instruction. When the iterator has traversed all the assigned instructions, it prepares to communicate the partial results back to the server node. The `evaluation_order` file was previously generated by the Mars code, but is now generated by the newly introduced Venus subprogram. This program is described in section 5.1. The `evaluation_order_indices` file generated by Venus is a new addition to JupiterNCSM.

6.3 Checkpointing

This project lays the groundwork for performing significantly larger simulations than what have been attempted so far with JupiterNCSM. Such new capabilities are welcome additions since they allow the study of more complex nuclear systems and the use of larger model spaces. However, these new capabilities also mean that the code will need more computational power to achieve the desired results. This power is typically attained by allowing the simulations to run for a longer time, or by using several computer nodes at the same time. These methods therefore increase the risk for data loss due to, for example, power loss in the data center or hardware failure in one of the computer nodes. Such increased risks must be handled to safeguard against loss of computational results. This includes loss of researcher time and loss of HPC CPU core hour allocations.

¹In this case the MPI node with world rank 0 is treated as the server node.

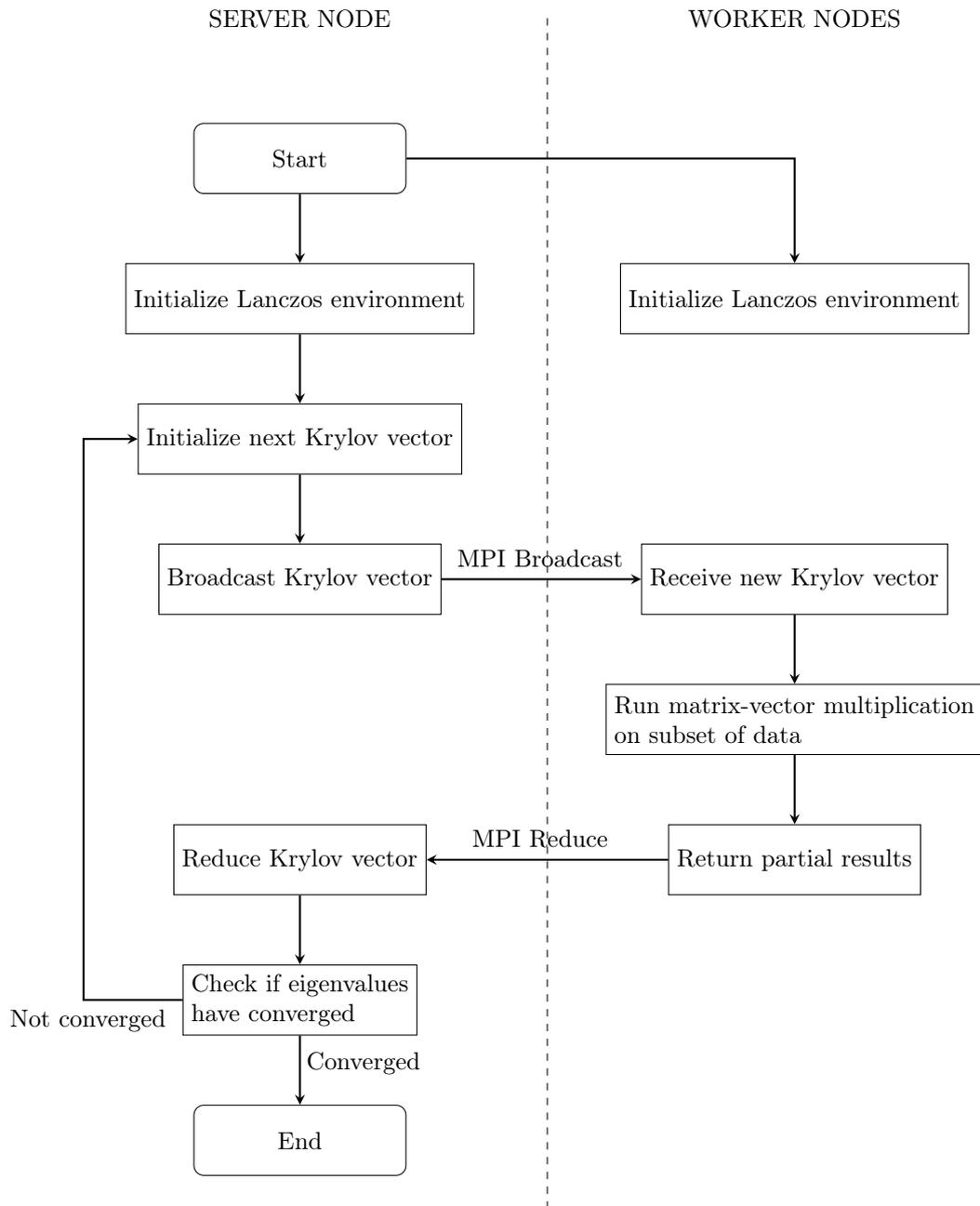


Figure 6.2: Flowchart of the MPI parallelization. The dashed line indicates the separation between the server node and the worker nodes. The processes that cross this barrier are implemented using MPI. Note that the server node is just a regular worker node that has been given a few extra tasks.

One way to mitigate these risks is to introduce checkpointing to the simulations. When the simulation reaches one of these checkpoints, it saves the current state of the system. If the program is unable to run to completion for some reason, then it is possible to restart from the latest checkpoint. Some progress is inevitably going to be lost, but a large part of the simulation can still be salvaged if the checkpointing and resume procedures are adequate.

The first version of the Lanczos solver Bacchus had no checkpointing feature. Since the goal of this project is to run simulations that are much larger than what has previously been attempted, a proper checkpointing and restart feature is needed. This feature was developed and introduced during the course of this project.

In order to be able to restart the Lanczos algorithm, we need access to the Krylov vectors and the tridiagonal matrix as defined in algorithm 1. The Krylov vectors are always saved to disk, so we only need to make sure the tridiagonal matrix is also saved continuously. Internally, this matrix is represented by two arrays of double precision floating point numbers. These arrays are then saved in a directory on disk together with all other data (Krylov vectors, index lists, etc.).

The resume feature is implemented by reading the tridiagonal matrix elements and Krylov vectors from disk to recreate their internal equivalents. Together with the iteration index, which is provided by the file name of the tridiagonal matrix elements, this is enough to completely recreate the internal state of the Lanczos solver at this checkpoint. From this state the solver is then able to continue as if nothing had happened. The Krylov vectors and tridiagonal matrix elements are saved after each iteration, so it is possible to restart the algorithm from any previous iteration, if so desired. The implementation works in both the single-node and multi-node configurations. In the multi-node case, the server node is responsible for reading the tridiagonal matrix from disk. It is possible to use a different number of nodes when restarting, compared to the original count.

Practically, this is implemented as an additional command line flag that is passed to Bacchus, the `--resume` flag. The default behavior if no additional argument is provided is to restart from the latest available iteration. It is also possible to choose which iteration to restart from by passing a positive integer after the flag. For example, `--resume 3` instructs Bacchus to restart from the third iteration.

Part III
Results

Chapter 7

Computational performance

As we have established in earlier chapters, using the NCSM poses significant computational challenges. JupiterNCSM is designed to handle the very large matrices that arise when solving the Schrödinger equation using this method, but even so, we are pushing the limits of what this tool is capable of.

To get the most value out of the software, we must make sure that it is indeed able to run the large simulations we want. A feasibility study is conducted in section 7.1. This is expanded in section 7.2, where run-time considerations are explored to make sure we can run our simulations in a reasonable amount of time. Once we have established which cases we should be able to run, we carry out performance profiling on JupiterNCSM to identify and improve components that are not working optimally. This is done in section 7.3. Results from the work-order performance optimization will be explored in section 7.4. Similarly, results from the distributed-memory parallelization will be explored in section 7.5.

Due to the nature of software performance profiling, some of the results presented in this chapter are specific to the JupiterNCSM software running on Tetralith hardware. Results will probably look somewhat different on other systems, but the overall behavior of the system should remain more or less the same. A summary of the technical specifications of typical Tetralith nodes is presented in table 7.1. All simulations are performed with 32 OpenMP threads and an energy eigenvalue convergence tolerance of 1×10^{-7} MeV, unless otherwise noted. For reference, In order to achieve convergence with this tolerance we need 21 Lanczos iterations for the $N_{\max} = 0$ case, up to 45 iterations for the $N_{\max} = 8$ case. Results presented in section 7.2 onwards are for the Lanczos solver Bacchus only. Time required for generation of matrix elements and similar is not included in the presented results.

7.1 Feasibility of ^8Be simulations at larger model spaces

As stated earlier, one of the main computational difficulties in this project relates to storing the Hamiltonian matrix. The dimensionality of this matrix increases rapidly as the number of nucleons, or the size of the model space, increases. To illustrate this, we see in figure 7.1 the total file size of the implicit matrix element data as a function of N_{\max} for three different nuclei, ^4He , ^6Li and ^8Be . The figure corresponds to the case with both 2NFs and 3NFs in the nuclear Hamiltonian. The nuclei ^4He and ^6Li have been studied before with JupiterNCSM [6], while ^8Be has not. Recall that simulating ^8Be at $N_{\max} = 8$ is one of the main goals in this project. As can be seen from the figure, the implicit matrix data for all cases with $N_{\max} \leq 6$ fits within the available RAM of a regular thin node. At $N_{\max} = 8$ we need to use a fat node to store the entire

Table 7.1: Summary of computational hardware on the Tetralith HPC system [15]. In this work we use thin, large disk nodes for profiling measurements. Fat nodes are also used when transforming matrix elements from J- scheme to M-scheme using Mercury, but performance in this process is not measured.

Node type	Number of nodes	CPU	CPU cores	RAM	Scratch disk
Thin	1674	2 × Intel Xeon Gold 6130, (2.1 GHz)	32	96 GiB	240 GB SSD
Thin, (large disk)	170	2 × Intel Xeon Gold 6130, (2.1 GHz)	32	96 GiB	2 TB NVMe
Fat, (medium disk)	60	2 × Intel Xeon Gold 6130, (2.1 GHz)	32	384 GiB	960 GB SSD
Fat	4	2 × Intel Xeon Gold 6130, (2.1 GHz)	32	384 GiB	240 GB SSD

^8Be input data in RAM. If we want to proceed to $N_{\max} = 10$ using only RAM, then we need significantly more memory than what is available, even on fat nodes.

Fortunately, JupiterNCSM can store the implicit matrix element data on secondary memory, such as HDDs or SSDs. If the data can be stored on disk, then only a small fraction of this, the current working set, needs to be loaded into RAM for processing. We can then focus on the dashed lines in figure 7.1, which indicate the amount of scratch disk storage available on the Tetralith nodes. The large disk nodes, each with a 2 TB NVMe disk [15], are then able to run almost all cases in the figure, except for the ^8Be , $N_{\max} = 10$ case which is just above the total available storage space.

In the future it would be beneficial to run the $N_{\max} = 10$ case for ^8Be , see chapter 10. In order to do this, we need to find a way to perform the simulation, despite the storage space limitation. One possible solution is to store the implicit matrix data on network attached disks. A very large file system can then be emulated, the size of which is mainly limited by financial considerations. This would solve the data size problem. Unfortunately, network storage is generally not as fast as node-local scratch storage. Using it would lead to a performance penalty, slowing down the entire simulation. In the case of the Tetralith cluster, this slowdown is on the order of a factor three, see section 3.3.

Another way of running the $N_{\max} = 10$ case is to divide the implicit matrix data between several nodes, see section 5.2. Not all nodes need access to the entire implicit matrix. We could assign one part of the Hamiltonian matrix to the first node, another part to the second node, etc. until we have used the combined storage capacity to store the entire set of implicit matrix data. This procedure is tested experimentally in the ^8Be , $N_{\max} = 8$ case in section 7.5. It works as expected and should scale well to the $N_{\max} = 10$ case.

In summary, ^8Be simulations at $N_{\max} = 8$ are indeed manageable in terms of storage space on a single node. It is also clear that simulations at $N_{\max} = 10$ pose significant additional challenges in terms of data management. We would then need large network storage to hold all implicit matrix data in one place, and two large disk nodes to actually perform the simulation from node-local scratch disks. The two nodes would need to use the file division scheme introduced in section 5.2.

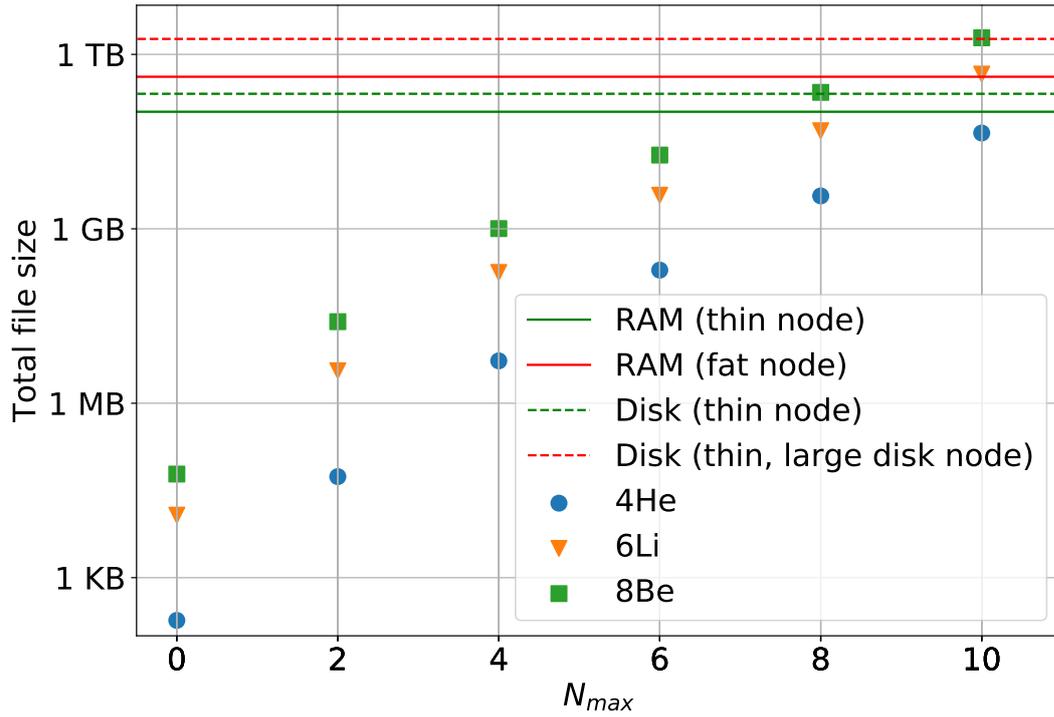


Figure 7.1: Total size of the JupiterNCSM implicit matrix data as a function of N_{max} for three nuclei, ^4He , ^6Li and ^8Be . The simulations are performed with both 2NFs and 3NFs in the nuclear Hamiltonian. At the top of the figure we have solid lines indicating the amount of RAM installed on Tetralith compute nodes. Additionally, the dashed lines indicate the amount of scratch disk storage available on the nodes [15].

7.2 Typical run times for the Lanczos solver

Now that we have established that it is indeed feasible to run the ${}^8\text{Be}$, $N_{\text{max}} = 8$ simulations on the computational hardware we have at our disposal, see section 7.1, we must consider if we are able to run it in a reasonable amount of time.

At this point we need to categorize our usage of JupiterNCSM into one of two classes. The first type of usage consists of solving a given many-body problem once. In this case we need to run each step in the processing chain once. In the second type of usage we solve the same many-body problem, but using several different Hamiltonians. In this case we need to run the pre-processing steps only once, but the Lanczos solver Bacchus must be run once for each Hamiltonian. Both the pre-processing steps and the Lanczos algorithm take significant amounts of time to complete. It is then clear that the total run-time requirements in the second case are heavily dependent on the number of Hamiltonians used. In this project we are mainly interested in this second type of usage, see for example the study in chapter 8, so run-time is going to be dominated by the Lanczos solver Bacchus. In the rest of this chapter we focus on run-time performance of Bacchus.

Please consider the “Mars evaluation order” in figure 7.2, which shows typical run times for a set of single-node Bacchus runs for ${}^8\text{Be}$. The code was run on a Tetralith thin, large disk node with 32 threads, see table 7.1. There is no problem in running the simulations with lower N_{max} values. For example, the $N_{\text{max}} = 6$ simulation takes about 23 minutes. But simulation run times start to become noticeable at $N_{\text{max}} = 8$, where the simulation takes about 19 hours. It is also clear from the figure that the step from $N_{\text{max}} = 6$ to $N_{\text{max}} = 8$ causes a sudden change in the slope of the graph. This new, steeper slope is caused by the fact that the matrix data no longer fits in RAM. JupiterNCSM must then fetch files from disk instead of from RAM file cache, which slows down the simulation considerably.

We want to pave the way for running the $N_{\text{max}} = 10$ case in the future, so we perform a rough extrapolation of the expected run time. The main factor limiting computational performance is disk data throughput rate, see section 7.3. When simulating ${}^8\text{Be}$, $N_{\text{max}} = 8$ on Tetralith, the average read speed is approximately 0.125 GiB/s when using network attached disks and 0.44 GiB/s when using node-local NVMe scratch disks. If we assume that this transfer rate is approximately the same for both the $N_{\text{max}} = 8$ and $N_{\text{max}} = 10$ cases, then we can translate the increased data demands of the $N_{\text{max}} = 10$ case directly into an increase in run time.

Using the disk read estimation tool described in section 5.3, we can estimate that the ${}^8\text{Be}$, $N_{\text{max}} = 10$ case would need to read about 15 TB of data from disk in every Lanczos iteration. A full simulation often needs on the order of 40 to 50 iterations, meaning that we need to read approximately 600 TB from disk. This corresponds to run times on the order of two to eight weeks, depending on if we use scratch-disk or network storage. The two-week scratch-disk estimate is shown as an open circle in figure 7.2.

The long expected simulation run time for the $N_{\text{max}} = 10$ case is a very pressing issue. For example, the maximum job time limit on Tetralith is one week [26]. It might be possible to get special permission to run a single job for longer than that, but it would not be practical. It is then clear that we need to optimize and parallelize the code using distributed memory if we want to run the $N_{\text{max}} = 10$ case in the future. See section 7.4 for results from the work-order performance optimization and section 7.5 for results from the distributed-memory parallelization.

The optimized work order introduced in section 5.1 reduces the ${}^8\text{Be}$, $N_{\text{max}} = 10$ data read requirements to about 2 TB per Lanczos iteration (estimated using the program in section 5.3). This is close to the total size of the matrix data, see figure 7.1. Assuming that we need 40 to 50 Lanczos iterations, this gives us a total data read requirement of about 100 TB. The simulation should then take between three and nine days on a single computer node, which is much more practical than the several weeks needed when using the old work order. The three-day estimation

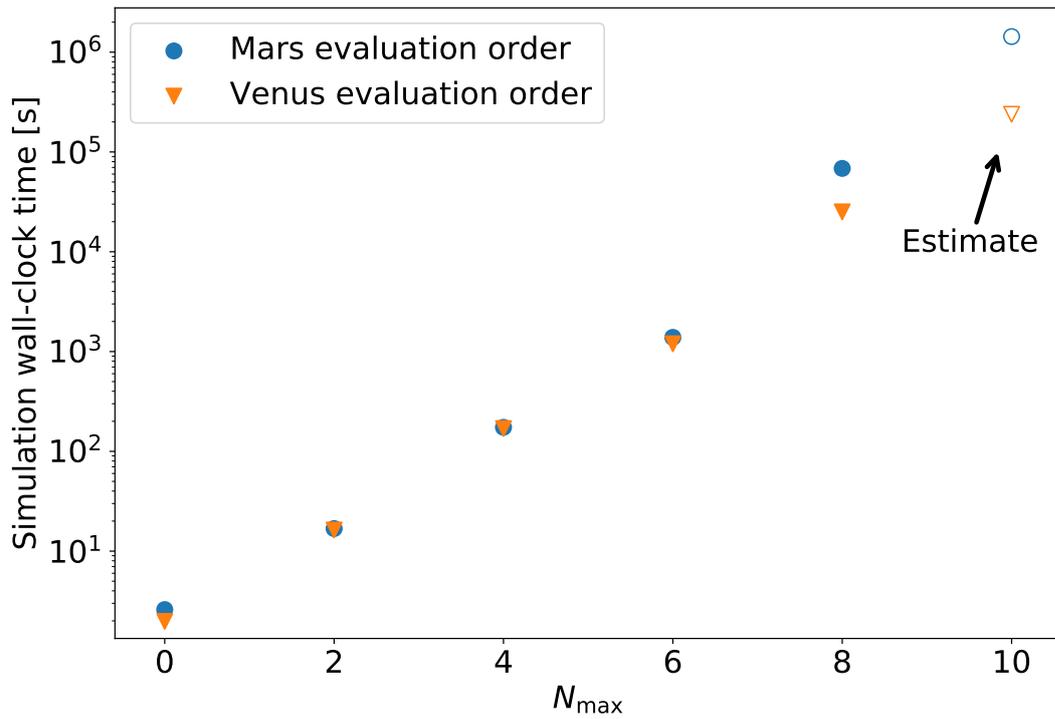


Figure 7.2: Total run time for the Lanczos algorithm in Bacchus as a function of N_{\max} for ${}^8\text{Be}$ using two different evaluation orders. The old version was computed using the Mars program in JupiterNCSM. The new, optimized, version of the evaluation order is computed by the new program Venus. All measurements, and the two estimates, refer to JupiterNCSM running on a single Tetralith thin, large disk node. The optimized evaluation order becomes important when the amount of data exceeds the size of RAM, which for the Tetralith thin nodes happens for $N_{\max} \geq 8$, see table 7.1. At $N_{\max} = 8$ we see a reduction in run time of almost a factor three. The figure also includes estimates for the $N_{\max} = 10$ case based on results from the disk load estimation program that was introduced in section 5.3.

is illustrated with an open triangle in figure 7.2.

7.3 Performance profiling of the Lanczos solver Bacchus

We begin by looking at the case ^8Be with $N_{\text{max}} = 8$, one of the largest cases that we are able to run on a single Tetralith thin, large disk node. We do this first for the single-threaded case. In figure 7.3, we show a bar diagram indicating the time spent waiting on the two major bottlenecks that were identified in the simulation, I/O (input/output) and CPU processing. The I/O bar refers to time spent waiting for implicit matrix data to be loaded from disk to RAM. Note that this simulation is run on a Tetralith thin, large disk node, see table 7.1, which means that not all data fits in RAM. Bacchus must then frequently perform I/O tasks. The time it needs for these operations is measured and presented in the figure. The CPU bar refers to time spent waiting for the CPU to finish processing matrix data. The results indicate that we are mostly bottlenecked by the CPU in the single-threaded case. But since this is only a single thread, we must be careful when generalizing to the multi-threaded case. We can get an idea of how the multi-threaded case will behave by also considering the utilization of the disk and CPU components of a single compute node, see figure 7.4. The node has a single disk, represented by the block on the left, and a CPU with total of 32 CPU cores, indicated by the 32 smaller blocks to the right. Note that the real compute node has two CPUs, each with 16 CPU cores. But for our purposes this can be considered to be equivalent to a single CPU with 32 cores. Regions colored red indicate that the resource is busy, green that it is idle. A resource can be partially filled with red, meaning that it is busy only a certain fraction of the time.

From the measurement presented in figure 7.3 we see that the disk is the active component about 20% of the time, whereas the CPU is active during the remaining 80% of the time. We can then draw the conclusion that disk (CPU) utilization is approximately 20% (80%) of total capacity, on average.

The single-threaded case is simple to reason about, but it is not very computationally efficient. To increase performance we go to the multi-threaded case. Since disk storage is a shared resource, there is an upper limit for how many CPU cores the disk can provide data to in a timely manner before data read latency increases significantly. We know that the disk is active about 20% of the time in the single-threaded case. We would then expect that it is able to continuously supply data to about 4 or 5 CPU cores. This has been experimentally tested and was found to be approximately true on average.

This situation is illustrated schematically in figure 7.5. Here, disk is utilized to full capacity, whereas only a few CPU cores are busy. Most of them are idle, waiting for data from disk. We are then only able to use a small fraction of the 32 available cores, which is not very efficient. From this reasoning it is clear that disk data throughput rate is the main bottleneck in the multi-threaded case when using the Lanczos solver Bacchus.

7.4 Optimized work order

A natural follow-up question is then if the effect of this data transfer bottleneck can be reduced. There are essentially two ways forward: we can either increase the data bandwidth between disk and main memory, or we can reduce the amount of I/O required in the simulation. If we can improve either (or both) of these two aspects, then we also increase CPU utilization, and consequently reduce simulation run times. In this project we mostly use fast NVMe disks, so increasing storage system performance is difficult. A possible alternative to the single-disk case would be to use several NVMe scratch disks in a RAID (redundant array of independent disks)

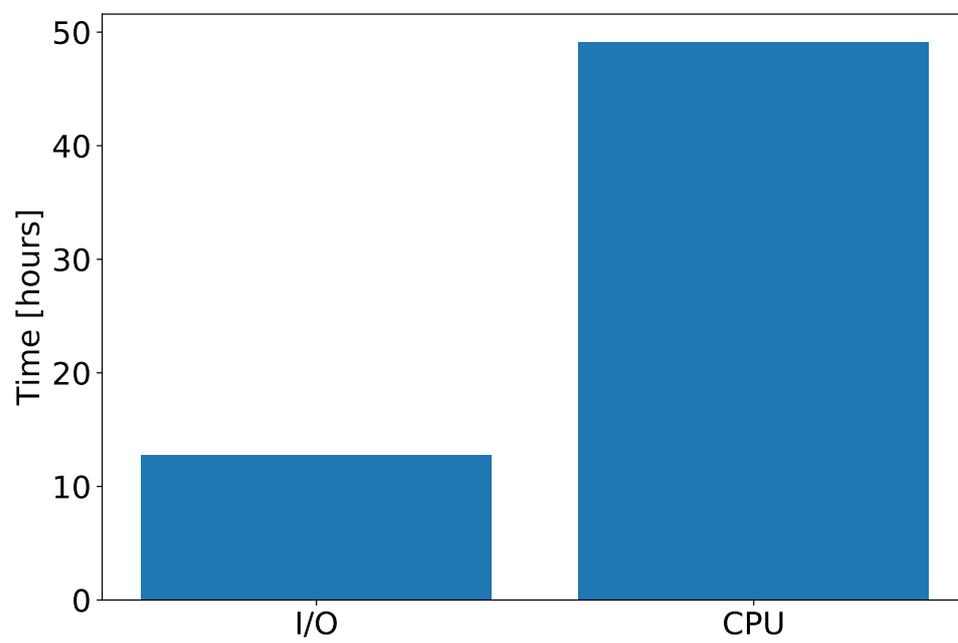


Figure 7.3: Time spent waiting for I/O and CPU. The simulation is for ${}^8\text{Be}$ at $N_{\text{max}} = 8$. Notice that this simulation is for the single-threaded case.

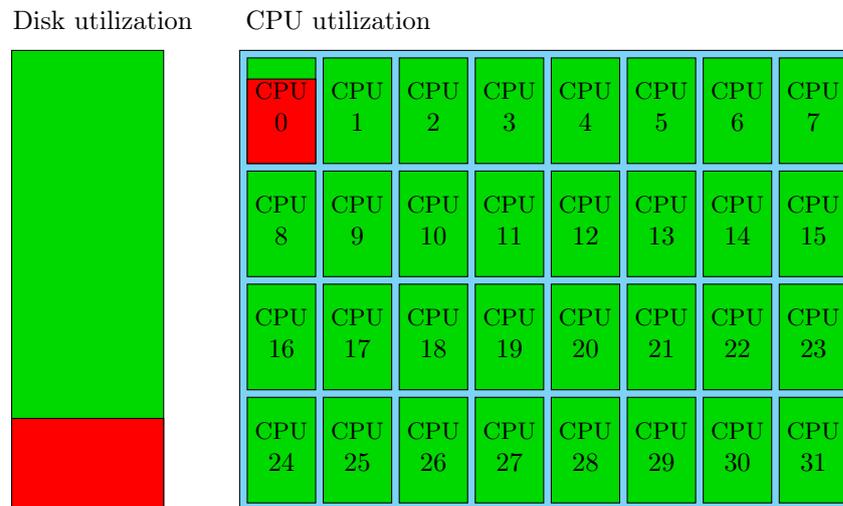


Figure 7.4: Schematic illustration of the disk and CPU utilization of a compute node running JupiterNCSM in a single thread. Red indicates that the resource is busy, green that it is idle. A resource that is partially filled with red indicates that it is busy only a fraction of the time. For example, the disk resource is approximately 20% filled with red, indicating that disk throughput is used to about 20% of total capacity.

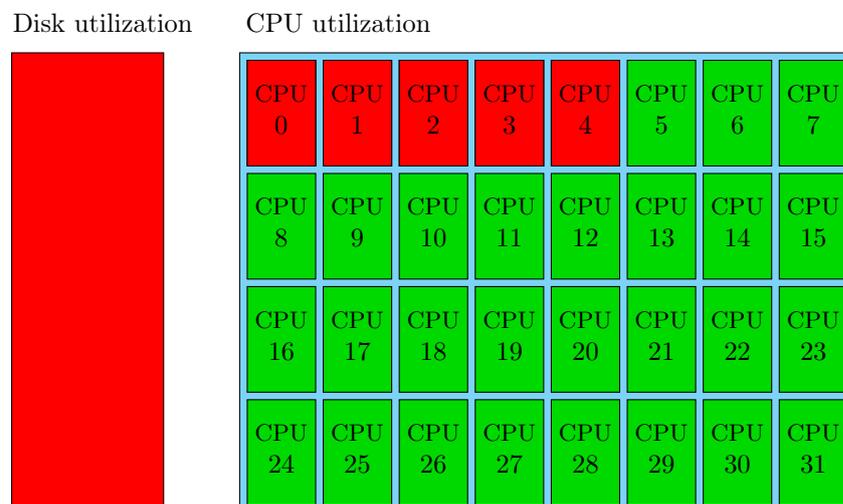


Figure 7.5: Multi-threaded equivalent of figure 7.4. Full utilization of disk data transfer rate is enough to continuously supply about 4 or 5 CPU cores with data. The other CPU cores are idle, waiting for data.

configuration to increase read throughput, but that option is typically not available on general HPC systems. We must then look at the second solution, to reduce the amount of I/O needed.

When using the Lanczos method to estimate eigenvalues to the Hamiltonian matrix, Bacchus must read the complete set of implicit matrix data at least once. In figure 7.6 we see a blue bar on the left (labeled “Old version”) representing the amount of data actually read from disk in a single Lanczos iteration for the case ${}^8\text{Be}$, $N_{\text{max}} = 8$. The dashed line close to 220 GB indicates the actual size of the matrix data, and therefore represents a theoretical lower limit on how much data must be read and processed in each iteration. The lower dashed line at 96 GiB \approx 103 GB indicates the amount of RAM installed on a typical compute node at Tetralith. It is clear that the input data does not fit in RAM, an important factor for performance. If the entire set of matrix data were to fit in RAM, then we would only need to read the data once in the entire simulation. But RAM is smaller than the data, which means that the matrix elements need to first be loaded into RAM, and then discarded after a while to make space for new data. The matrix elements are used several times in each Lanczos iteration, which means that the same data might be read more than once. This is a rather inefficient use of resources, but unfortunately a consequence of the large data sizes.

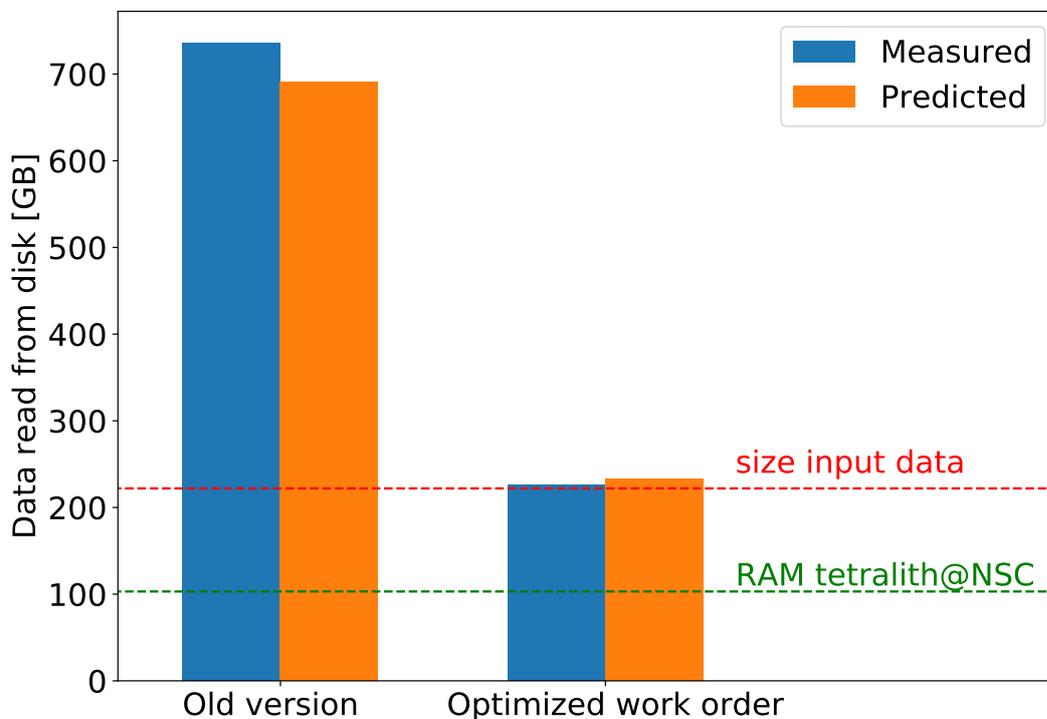


Figure 7.6: Amount of data read per Lanczos iteration for the old (Mars) and the optimized (Venus) work order for a simulation of ${}^8\text{Be}$ at $N_{\text{max}} = 8$. The upper dashed line indicates the size of the implicit matrix data. The lower dashed line indicates the amount of RAM on thin compute nodes at Tetralith. The figure also shows predicted results from the disk estimation tool introduced in section 5.3.

Importantly, these measurements show that Bacchus, when using the old evaluation order

computed using Mars, reads much more data than the approximately 220 GB that is the total size of the implicit matrix data. For example, in the ${}^8\text{Be}$, $N_{\text{max}} = 8$ case illustrated in figure 7.6, Bacchus reads the entire input data set more than three times in each iteration, on average. This is, of course, not very efficient. The work order, and therefore file access pattern, can to some degree be adjusted. One possibility for reducing the amount of data that must be re-read from disk involves sorting the files by size and then processing them in decreasing order, starting with the largest file as discussed in section 5.1. Implementing this optimized work order is an important result of this work. It turned out that this new file access pattern could significantly decrease the number of times a file had to be re-loaded into RAM, see the blue “Optimized work order” bar in figure 7.6. We can see from the figure that the new file access pattern significantly reduces the amount of data that must be read. It is reduced essentially to the theoretical lower limit, a more than factor three improvement. Since Bacchus is heavily bottlenecked by disk throughput rate, it means that this improvement translates almost directly into an equivalent increase in overall computational performance, as can be seen in figure 7.2.

In figure 7.6 we also see two orange bars representing the disk load predictions from the disk load estimation tool introduced in section 5.3. The predictions are quite close to the real measured values, which indicates that this tool has good predictive power and thereby can remove the need for expensive measurements. If we also have average disk read rates when running Bacchus, such as the ones presented in section 3.3, then we can predict total simulation run times with good accuracy. This type of prediction is useful when assessing the feasibility of completing a given simulation within a certain time frame.

A measurement of how this optimization affects the simulation run time is shown in figure 7.2. This figure shows the total run time for the Lanczos algorithm in Bacchus as a function of N_{max} for the old (Mars) and the optimized (Venus) work order. The simulations are for ${}^8\text{Be}$ on a single thin, large disk node on Tetralith. The run time of simulations with $N_{\text{max}} \leq 6$ are not affected by the new evaluation order. This is because the entire set of implicit matrix data fits in RAM. It stays there throughout the entire simulation, meaning that there is no need to read data from disk. It then eliminates disk data throughput rate as the main performance bottleneck.

Conversely, the $N_{\text{max}} = 8$ case is clearly affected by the new evaluation order. The increased size of the matrix data means that it no longer fits in RAM, so it must be read from secondary storage. Disk throughput is then the major performance bottleneck, and a reduction in the amount of read data in each iteration translates almost directly into an equivalent reduction in simulation run time. Here we observe a reduction of almost a factor three in time required to run the Lanczos algorithm when using the optimized work order. A prediction for the $N_{\text{max}} = 10$ case is also provided, and the effect of the new work-order is even more noticeable.

7.5 MPI performance scaling

So far in this chapter we have mainly treated the single-node computational performance of JupiterNCSM. The introduction of a distributed-memory architecture using MPI, one of the main achievements of this project, means that the software is parallelized beyond the single-node OpenMP parallelism. This allows computational performance to scale more efficiently.

Figure 7.7 shows the Bacchus simulation run time required for the ${}^8\text{Be}$, $N_{\text{max}} = 8$ case as a function of the number of MPI compute nodes when using JupiterNCSM. The NVMe times include time for copying data from network to node-local storage, an operation that is not needed when using network storage. The copy takes about 40 minutes in the single-node case, less in the multi-node case because we are using the file division scheme from section 5.2. For reference, the copy time when using eight nodes is about 14 minutes.

As expected, the required run time decreases as more nodes are used. Using more MPI nodes means that we achieve a higher data throughput rate, which also decreases simulation run time. It is also expected to see that the advantage of using NVMe disks decreases as more nodes are used. This is because using more nodes means that we have a larger total RAM space. More implicit matrix data can then be stored in fast RAM file cache, which decreases the need for slow storage file access.

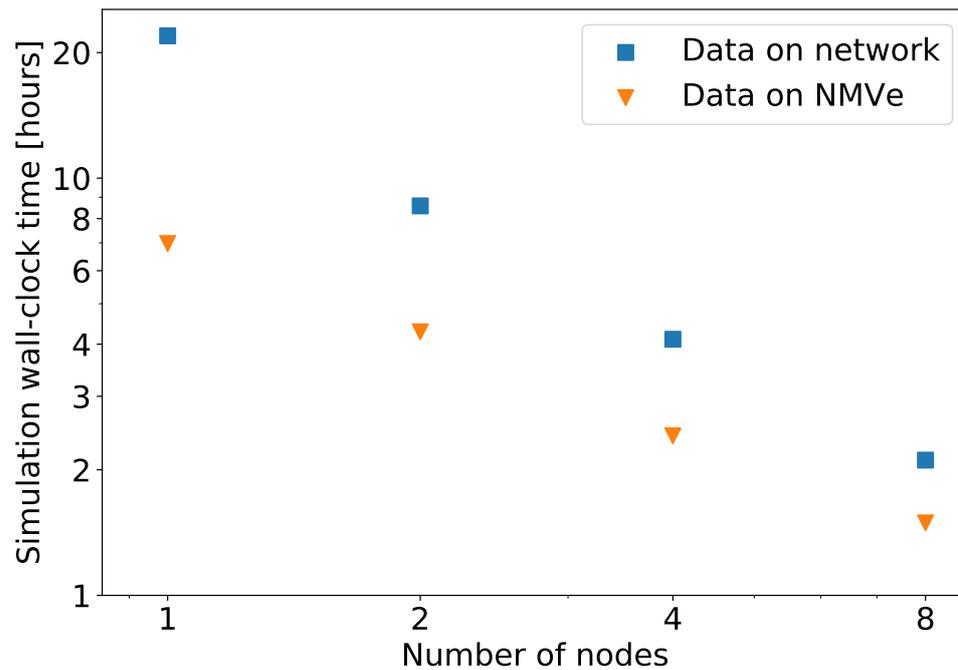


Figure 7.7: Simulation run time for the ${}^8\text{Be}$, $N_{\text{max}} = 8$ case as a function of the number of Tetralith MPI nodes. The figure shows two different cases, one where implicit matrix data is stored on network-attached disks, the other where matrix data is stored on node-local NVMe scratch disks.

Chapter 8

Statistical study of the ${}^8\text{Be}$ decay energy threshold

The results presented so far have been on the computational performance of the JupiterNCSM code, with particular focus on the performance achieved on the Tetralith HPC system. However, the main purpose of this research code is to study nuclear many-body systems and learn about the nuclear interaction. In this section we explore the physics results that came out of this project.

Since JupiterNCSM has not been used for simulating ${}^8\text{Be}$ before, we start by validating it against a previously developed NCSM code in section 8.1. With the tool validated, we study the NCSM ground-state energy convergence rate as a function of N_{max} in section 8.2. Using extrapolation formulas for the convergence rates, we can then study how statistical uncertainties in two of the parameters in the chiral interaction model employed in this work affect the computed ground-state energy of ${}^8\text{Be}$. This is then used to compute the ${}^8\text{Be}$ decay energy threshold, which is the energy difference between the ground states of ${}^8\text{Be}$ and two ${}^4\text{He}$ nuclei.

8.1 Validation of ${}^8\text{Be}$ results

We begin by validating new JupiterNCSM results for ${}^8\text{Be}$ against results from pAntoine, which is a previously developed NCSM code that only handles 2NFs [14]. Figure 8.1 shows the computed ${}^8\text{Be}$ ground-state energy as a function of the basis truncation parameter N_{max} . The simulations use the NNLO_{sat} interaction with a harmonic oscillator basis frequency of $\hbar\omega = 20 \text{ MeV}$ [27]. Validation energies are calculated using pAntoine. The absolute difference between the pAntoine and JupiterNCSM 2NF results are $3 \times 10^{-5} \text{ MeV}$ or less. This is illustrated in the figure with an almost complete overlap between the results. From this we conclude that JupiterNCSM is able to accurately reproduce these 2NF results. The small difference is attributed to the use of slightly different convergence criteria in the two codes. JupiterNCSM assumes that it has reached convergence when the absolute difference in the lowest eigenvalue between two consecutive Lanczos iterations is smaller than $1 \times 10^{-7} \text{ MeV}$. The equivalent limit in pAntoine is $1 \times 10^{-4} \text{ MeV}$. The observed difference of $3 \times 10^{-5} \text{ MeV}$ means that it can be attributed to the (larger) pAntoine convergence criterium. Figure 8.1 also shows first 2NF+3NF results from JupiterNCSM for the ${}^8\text{Be}$ nucleus with the NNLO_{sat} interaction. As can be seen in the figure, the convergence rate is lower in the 2NF+3NF case compared to the 2NF-only case.

From this analysis we can see that JupiterNCSM is able to reproduce 2NF-only results from

pAntoine. JupiterNCSM has, for smaller nuclei, been validated in the 2NF+3NF case against the nsopt software [28, 2]. We are, therefore, confident that JupiterNCSM works as expected for the full 2NF+3NF case.

8.2 ^8Be convergence study

From figure 8.1 it is clear that the computed ground-state energy E_{gs} for ^8Be converges towards a fixed value as N_{max} increases. The convergence behavior of the ground-state energy can be approximated using an exponential function on the form [6]

$$E(N_{\text{max}}) = E_{\infty} + a \exp(-bN_{\text{max}}), \quad (8.1)$$

where E_{∞} is an estimator for the ground-state energy of the fully converged NCSM solution as $N_{\text{max}} \rightarrow \infty$. The parameters a and b describe the convergence rate. These three parameters can be found for a particular nuclear state by fitting the exponential function to the computed ground-state energy for a few different N_{max} values. Figure 8.2 shows a graphical representation of this procedure for the two cases, 2NF-only and 2NF+3NF, for a ^8Be nucleus using a chiral interaction model as defined in [6] and LEC values from [7]. Using these interactions, the fitted values for E_{∞} are -59.77 MeV and -54.93 MeV for the 2NF and 2NF+3NF case, respectively.

If we had been able to run NCSM simulations for $N_{\text{max}} \rightarrow \infty$, then the fully converged NCSM ground-state energy could be denoted $E(c_D, c_E)$, where c_D and c_E are LECs in the nuclear Hamiltonian, see equation (1.3). Our simulations are, unfortunately, limited to finite N_{max} . A simulation would then give us $E_{\text{NCSM}}(c_D, c_E, N_{\text{max}})$. In order to get an approximation for the $N_{\text{max}} \rightarrow \infty$ case, we need to add a method-error correction term $\delta E(c_D, c_E, N_{\text{max}})$. This error term is a stochastic variable that will be more precisely defined in section 8.3. In summary, this gives us [6]

$$E(c_D, c_E) \approx E_{\text{NCSM}}(c_D, c_E, N_{\text{max}}) + \delta E_{\text{NCSM}}(c_D, c_E, N_{\text{max}}). \quad (8.2)$$

An important consideration to keep in mind is that equation (8.1) is just an empirical model. The real convergence behavior can be somewhat different. This means that the prediction E_{∞} has a finite precision, with the uncertainty growing with the distance we need to extrapolate. We define the convergence distance as

$$\Delta E_{\infty}(c_D, c_E, N_{\text{max}}) \equiv E_{\infty}(c_D, c_E) - E(c_D, c_E, N_{\text{max}}). \quad (8.3)$$

This quantity is an estimation of how far away an NCSM solution at finite truncation N_{max} is from the fully converged result. Extrapolations performed with small $\Delta E_{\infty}(N_{\text{max}})$ have smaller errors than extrapolations with large $\Delta E_{\infty}(N_{\text{max}})$.

8.3 Three nucleon force uncertainty study

The chiral interaction model employed in this work has 16 parameters [7]. In particular, the strength of the 3NF interaction is parametrized by a pair of scalar LECs, see equation (1.3). The numerical values of the interaction parameter pair are only known through statistical inference from low-energy data, which means that we cannot use a single set of LECs to describe the interaction and make predictions for nuclei. Instead, the LEC values have to be sampled from the probability distributions, and then used in a statistical analysis.

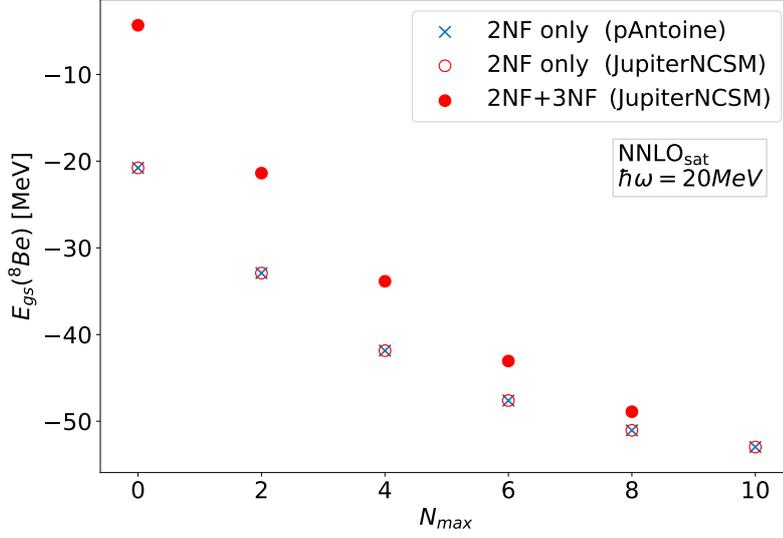


Figure 8.1: Validation runs comparing JupiterNCSM to the (2NF-only) pAntoine code. The absolute difference between pAntoine and JupiterNCSM (2NF only) is at most 3×10^{-5} MeV. This is close to the pAntoine convergence criteria of 1×10^{-4} MeV, which explains the small difference. The chiral interaction model used in this study is NNLO_{sat} [27] with and without 3NFs.

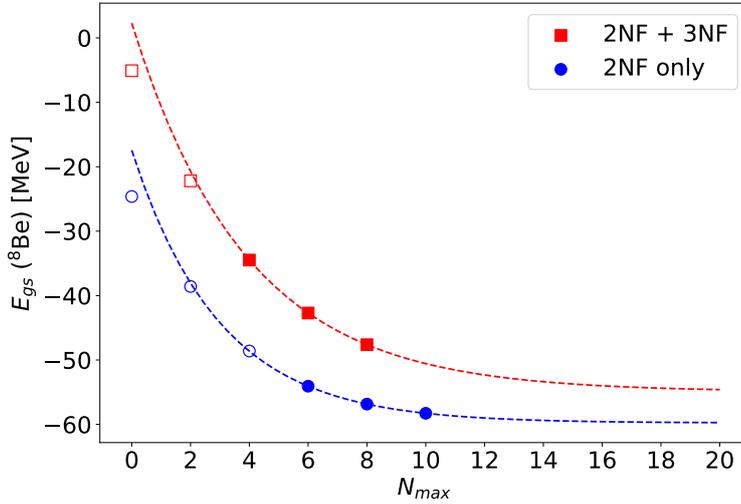


Figure 8.2: Ground-state energy E_{gs} as a function of N_{max} . Results for both 2NF-only and 2NF+3NF simulations using JupiterNCSM. These results were generated using interactions and LEC values as specified in [6] and [7]. The fitted values for E_{∞} were -59.77 MeV for the 2NF-only case and -54.93 MeV for the 2NF+3NF case. The fit and extrapolation were made using the three data points represented by filled boxes and circles in the figure.

In this study we draw 100 pairs of 3NF LEC values from the $\text{pr}(c_D, c_E|D, I)$ distribution obtained in [7] and then run an NCSM simulation for each pair. This gives us a set of LEC pairs and the corresponding NCSM ground-state energy.

The NCSM is a variational method, which means that it represents an upper bound to the fully converged NCSM ground-state energy. This is especially important when N_{max} is finite and the lowest energy eigenvalue has not yet converged to the infinite model space equivalent. The variational nature of this method means that the expectation value of the method-error correction term is negative, i.e. [6]

$$\text{E}[\delta E_{\text{NCSM}}(c_D, c_E, N_{\text{max}})] \equiv \mu_{\delta E_{\text{NCSM}}}(c_D, c_E, N_{\text{max}}) < 0. \quad (8.4)$$

The magnitude of this term is largely dependent on the convergence distance $\Delta E_{\infty}(c_D, c_E, N_{\text{max}})$ defined in equation (8.3). But since this distance is estimated on the basis of an extrapolation, it suffers from an extrapolation error σ_{NCSM} . Taken together, they give an approximation of the expectation value of the method error correction term, which is given by

$$\mu_{\delta E_{\text{NCSM}}}(c_D, c_E, N_{\text{max}}) = \Delta E_{\infty}(c_D, c_E) + \sigma_{\text{NCSM}}(c_D, c_E). \quad (8.5)$$

In [6], the authors mention that the extrapolation procedure often underestimates the extrapolation distance. They estimate the error in the extrapolation distance to

$$\sigma_{\text{NCSM}} = 0.2\Delta E_{\infty}(c_D, c_E). \quad (8.6)$$

However, they also note that this estimation is too large for the ^4He nucleus, so they assign it a numerical value of $\sigma_{\text{NCSM}} = 120 \text{ keV}$. Furthermore, they assume that the extrapolation errors are normally distributed, which means that the true value of the method error correction term is sampled from a normal distribution according to

$$\delta E_{\text{NCSM}}(c_D, c_E, N_{\text{max}}) \sim \mathcal{N}(\mu_{\delta E}(c_D, c_E, N_{\text{max}}), \sigma_{\text{NCSM}}^2). \quad (8.7)$$

The process of generating samples from the decay energy PPD $\text{pr}(Q_{2\alpha}|D, I)$ is summarized in algorithm 2. This process generates a total of $m \cdot n$ samples from this PPD, which can be described by

$$\text{PPD} = \text{pr}(E_{s_{\text{Be}}}|D, I) = \{E_{s_{\text{Be}}}(c_D, c_E) : c_D, c_E \sim \text{pr}(c_D, c_E|D, I)\}, \quad (8.8)$$

where D represents previous few-body data [7] and I denotes other information, such as models used and assumptions. The variable m represents the number of NCSM samples simulated and n represents the number of samples from the method error correction term for each NCSM sample. In this study we use $m = n = 100$. An equivalent analysis as the one for equation (8.8) can be done for ^4He , which gives us the PPD $\text{pr}(E_{^4\text{He}}|D, I)$.

We aim to study the decay energy threshold of ^8Be as it decays via the reaction



The energy released $Q_{2\alpha}$ in this reaction is defined as

$$Q_{2\alpha} = E_{\text{gs}}(^8\text{Be}) - 2E_{\text{gs}}(^4\text{He}) \quad (8.10)$$

This energy has been experimentally measured to be $Q_{2\alpha} \approx 92 \text{ keV}$ [29]. A graphical representation of this decay process is shown in figure 8.3. We wish to calculate $Q_{2\alpha}$ theoretically, but we do not have access to exact values of $E_{\text{gs}}(^8\text{Be})$ and $E_{\text{gs}}(^4\text{He})$. But we do have access

Algorithm 2: Schematic representation of the sampling process used in this chapter. This algorithm generates a total of $m \cdot n$ samples from the decay energy PPD $\text{pr}(Q_{2\alpha}|D, I)$.

```

1 for  $m$  do
2   Draw a pair of  $c_D$  and  $c_E$  values from the distribution  $\text{pr}(c_D, c_E|D, I)$ .
3   Solve the many-body Schrödinger equation with  $H = H(c_D, c_E)$ , see equation (1.3),
   using the NCSM. Do this for a few values of  $N_{\text{max}}$ . In this case we use
    $N_{\text{max}} = 4, 6, 8$ .
4   Apply the extrapolation formula to obtain (using equations (8.3) and (8.5)):
    $\Delta E_{\infty}(c_D, c_E, N_{\text{max}}) \equiv E_{\infty}(c_D, c_E) - E(c_D, c_E, N_{\text{max}})$ ,
    $\mu_{\delta E_{\text{NCSM}}}(c_D, c_E, N_{\text{max}}) = \Delta E_{\infty}(c_D, c_E) + \sigma_{\text{NCSM}}(c_D, c_E)$ .
5   if Adding extrapolation error then
6     Draw  $n$  samples  $\delta E_{\text{NCSM}}(c_D, c_E, N_{\text{max}}) \sim \mathcal{N}(\mu_{\delta E}(c_D, c_E, N_{\text{max}}), \sigma_{\text{NCSM}}^2)$ 
7      $E_{\text{sBe}}(c_D, c_E, N_{\text{max}}) = E_{\infty}(c_D, c_E) + \delta E_{\text{NCSM}}(c_D, c_E, N_{\text{max}})$ 
8   else
9      $E_{\text{sBe}}(c_D, c_E, N_{\text{max}}) = E_{\infty}(c_D, c_E) + \mu_{\delta E}(c_D, c_E, N_{\text{max}})$ 
10  end
11 end

```

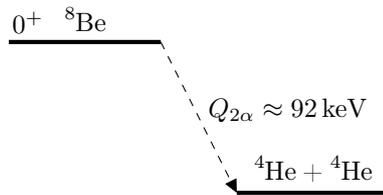


Figure 8.3: Schematic representation of ${}^8\text{Be}$ decay.

to the corresponding PPDs, $\text{pr}(E_{s_{\text{Be}}}|D, I)$ and $\text{pr}(E_{4_{\text{He}}}|D, I)$. We can then sample $Q_{2\alpha}$ using ground-state energy PPD samples drawn from $\text{pr}(E_{s_{\text{Be}}}|D, I)$ and $\text{pr}(E_{4_{\text{He}}}|D, I)$. Each sample pair drawn from $\text{pr}(E_{s_{\text{Be}}}|D, I)$ and $\text{pr}(E_{4_{\text{He}}}|D, I)$ must, of course, use the same pair of c_D and c_E values. We can then calculate the decay energy PPD as

$$\text{pr}(Q_{2\alpha}|D, I) = \{E_{s_{\text{Be}}}(c_D, c_E) - 2E_{4_{\text{He}}}(c_D, c_E) : c_D, c_E \sim \text{pr}(c_D, c_E|D, I)\}. \quad (8.11)$$

Figure 8.4 shows 100 samples drawn from the decay energy PPD, with $\sigma = 0$. The dashed vertical line indicates the experimentally measured decay energy. The distribution, with a standard deviation of about 0.3 MeV, captures the experimental value, which is encouraging. However, this width is still quite large, considering that the experimental value is only 92 keV.

To include effects of the uncertainties in the extrapolation procedure, we also sample the decay energy PPD with non-zero values of σ , as specified in equation (8.6). This is shown in figure 8.5, where we sample $\delta E_{\text{NCSM}}(c_D, c_E, N_{\text{max}})$ 100 times for each sample of $E_{\text{NCSM}}(c_D, c_E, N_{\text{max}})$. This gives us a total of $100 \cdot 100 = 10\,000$ samples of the decay energy PPD. In practice, this means that the distribution is widened to correspond to our uncertainty in the extrapolation procedure. The standard deviation of the distribution is in this case about 1.5 MeV, which is significantly wider than the $\sigma = 0$ version in figure 8.4. This increased width indicates that we have a large uncertainty in the extrapolation procedure. This can be reduced by decreasing the extrapolation distance, which requires NCSM simulations at larger values of N_{max} .

A more extensive error quantification analysis would also need to consider model errors, but that is beyond the scope of this project. However, this type of analysis is performed in [6] for $A = 6$ observables.

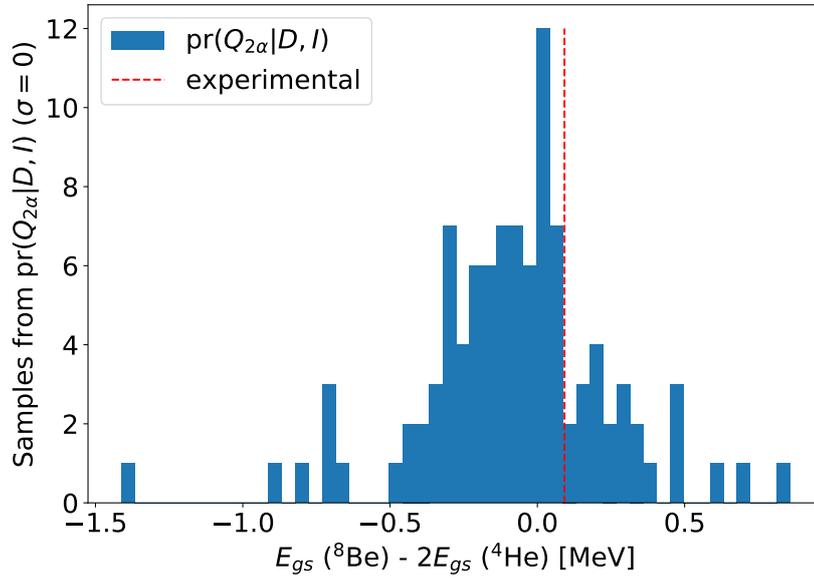


Figure 8.4: Sampling of the decay energy PPD, $\text{pr}(Q_{2\alpha}|D, I)$, using 100 samples. Notice that this figure does not include corrections for method uncertainty, i.e. the figure shows the $\sigma = 0$ case. The vertical dashed line indicates the experimentally measured energy difference [29].

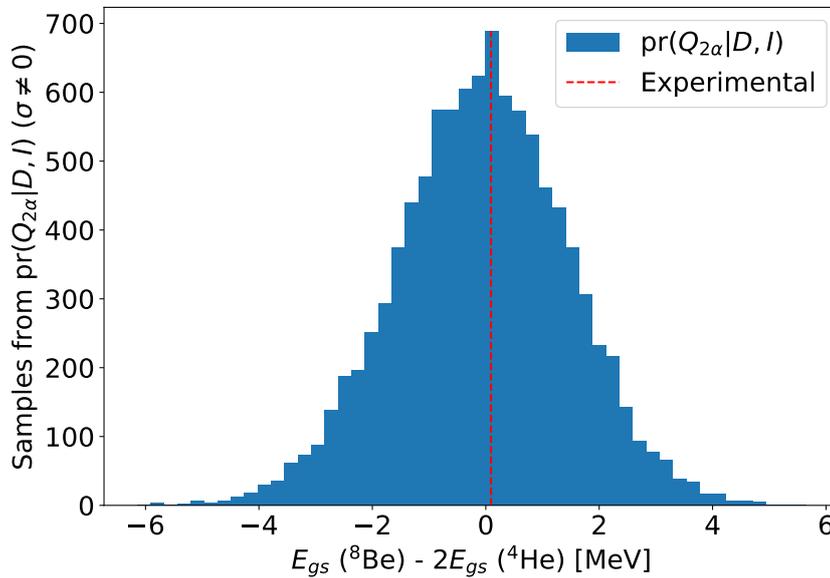


Figure 8.5: Sampling of the decay energy PPD, $\text{pr}(Q_{2\alpha}|D, I)$, using $100 \cdot 100 = 10000$ samples. This figure includes corrections for method uncertainty as specified in equation (8.6), i.e. $\sigma \neq 0$.

Part IV

Summary, discussion and outlook

Chapter 9

Summary

The overarching goal in this project has been to study the ground-state energy of the ^8Be nucleus starting from realistic nuclear interactions and solving the many-body Schrödinger equation using the NCSM. This type of simulation requires substantial amounts of computational power. A significant component of this project was focused on issues relating to the feasibility of performing large-scale NCSM calculations.

One of the investigation points related to evaluating the computational performance of the JupiterNCSM code. In particular, disk data throughput rate was found to be one of the major performance concerns. A distributed-memory architecture using MPI was then introduced to JupiterNCSM which allows the program to use several compute nodes at the same time. This increases parallelism and counteracts the disk bandwidth limitations by allowing the program to read more data in parallel. This new memory architecture increases code simulation capabilities and improves computational performance. Additionally, the impact of the limited data throughput rate was alleviated by finding a more optimal work order that maximizes reuse of data in RAM. Finally, NCSM simulations with large N_{max} need a lot of storage space. Only network storage is large enough for storing all matrix data in a single place, but it is typically slower than node-local scratch disks. To address this, a file division scheme was devised and implemented that allows JupiterNCSM to use a set of node-local scratch disks instead of network storage, avoiding a significant performance penalty for the large N_{max} cases.

With the performance improvements in place, we studied a few physical aspects of the ^8Be nucleus. A special case of the ^8Be simulations in JupiterNCSM was validated against another NCSM code, to verify that it works as expected. While performing this study we found that the computed ground-state energy was not fully converged for $N_{\text{max}} = 8$, the largest case that we were able to simulate. To compensate for this, we implemented a previously developed procedure for extrapolating the computed ground-state energy as $N_{\text{max}} \rightarrow \infty$. This gave us an approximation to the result of the fully converged NCSM simulation for large N_{max} .

With this result, and with results from previous ^4He studies performed with JupiterNCSM, we studied the decay energy threshold of ^8Be as it decays into two ^4He nuclei. This decay energy is known from experiment, but it is also possible to compute it as the difference between the ground-state energies of the ^8Be nucleus and the two ^4He nuclei. These ground-state energies come from NCSM simulations and depend on, among others, the c_D and c_E LECs. The values of these constants are only known through probability distributions, so we needed to sample them. This means, by extension, that the computed decay energy threshold is a sampling of the real decay energy. This was presented as a histogram of NCSM energy differences.

Chapter 10

Discussion and outlook

We achieved the goal of simulating the ${}^8\text{Be}$ nucleus at $N_{\text{max}} = 8$. However, from the results presented in figure 8.2, it is clear that the NCSM calculations are not fully converged. The NCSM results with only 2NFs is relatively close to converging, as indicated by the shallow slope at $N_{\text{max}} = 10$. However, the simulation with both 2NFs and 3NFs is still some distance away from full convergence, as can be seen from the still quite significant slope at $N_{\text{max}} = 8$. Consequently, it would be desirable to run simulations including both 2NFs and 3NFs at $N_{\text{max}} = 10$ and possibly $N_{\text{max}} = 12$. This would decrease the extrapolation errors, which in turn would make the predictive distribution in figure 8.5 more narrow. We did attempt to run the ${}^8\text{Be}$ case with $N_{\text{max}} = 10$, but due to difficulties with large RAM requirements in the J- to M-scheme transformation code Mercury, we were unable to generate the needed implicit matrix data files.

We attempted to locate the root cause of this issue by applying typical debugging techniques. The idea was that if we could locate the point in the code where the out-of-memory crashes were triggered, then we would be able to determine what had caused them. Unfortunately, the program seemed to crash at different places each time. This made debugging quite challenging. One of the debugging approaches involved tracking the amount of RAM Mercury used, to see if the excessive RAM usage was constant or sudden. It appeared that Mercury respected the imposed memory limit most of the time, but sometimes it would suddenly spike in RAM usage, far above the specified limit. This meant that most of the crashes were due to the process being killed by the Linux out-of-memory killer. This happens when physical memory on the computer is exhausted. But there were also some Mercury runs that crashed due to segmentation faults. This type of crash happens when a process tries to access a memory address that it should not have access to. It is likely that these symptoms are related to the out-of-memory errors. It could, for example, be that a memory allocation using `malloc` fails. A segmentation fault would then be triggered when Mercury attempts to use that memory block. Currently, not all `malloc` calls are checked for a NULL return value. The segmentation fault might then be caused by the code trying to dereference a NULL pointer.

A possible workaround to the RAM issue could have been to use a node with even more RAM. There are computer nodes on other HPC systems that have 700-800 GiB of RAM or even more. Unfortunately, we did not have access to any of these nodes.

Another issue that made the $N_{\text{max}} = 10$ case even more difficult was that, even if we managed to solve the issue of high RAM usage, the transformation process was still very slow. In the case where Mercury crashed after three days, it had only managed to generate about 10% of the data that was needed for running the $N_{\text{max}} = 10$ case. It is expected that the data generation rate is more or less constant throughout the entire transformation process, meaning that the full

transformation would take about a month to complete.

The RAM issue, together with the slow progress rate in the transformation, makes it impractical to run the $N_{\max} = 10$ case at this time. It is worth noting, however, that these limitations are due to the Mercury code. The Lanczos solver Bacchus, which has been the focus of this project, should have no problem handling the $N_{\max} = 10$ case.

Some of the newly added performance improvements in JupiterNCSM might not have been strictly necessary for the ${}^8\text{Be}$, $N_{\max} = 8$ case studied in this project, but they will become crucial for future calculations with larger model spaces. For example, the work order optimization will have a very noticeable effect on any simulation where the implicit matrix data does not fit in RAM. This is already true for the ${}^8\text{Be}$, $N_{\max} = 8$ case, but the effect will be even more pronounced for larger values of N_{\max} . Also, the distributed-memory architecture introduced in this project means that significant increases in performance can be achieved by using several computer nodes at the same time, something that will be critical for our ability to run simulations with even larger N_{\max} .

10.1 Suggested improvements

Some of the studies and improvements done in this project were based on the outlook in the PhD thesis by Tor Djärv [2]. Other investigation points emerged and were studied as the project progressed. As a result, several aspects of the tool have been improved, especially in terms of computational performance. This allows the study of more complex nuclei at larger model spaces. However, there are still areas that need further development, with the most important ones outlined below.

Improve J- to M-scheme transformation. The main bottleneck preventing us from running larger N_{\max} cases is the J- to M-scheme transformation code Mercury. The current version requires more RAM than what is available, leading to crashes. Also, even if the cause of the crashes is fixed, the transformation routine is very inefficient and time-consuming. It would probably need a dedicated optimization effort to increase performance so that it can complete the transformation within a reasonable amount of time. This would probably require a complete rewrite of the program [30].

Simulate ${}^8\text{Be}$ at $N_{\max} = 10$ or higher. As was covered in the discussion, the $N_{\max} = 8$ simulations for ${}^8\text{Be}$ suffer from relatively large uncertainties from the extrapolation procedure. Simulations at $N_{\max} = 10$ or higher would decrease these uncertainties. This would give more useful calculations of ground-state energies and, as observed in section 8.3, would lead to higher theoretical precision for the ${}^8\text{Be}$ decay energy threshold. The required improvements to the Lanczos solver Bacchus have already been implemented. It only needs the transformed matrix elements generated by the Mercury code.

Break dependencies to other codes. The JupiterNCSM software has strong dependencies to external codes and data. These codes must be run independently, and the data they generate must be manually fed to even more programs outside the JupiterNCSM toolchain. This makes it difficult to use the software. One important improvement that is needed to make JupiterNCSM more user-friendly is to break these dependencies so that all the required abilities are included in the JupiterNCSM toolchain. This includes the generation of input data in the form of a many-body basis and possibly also the 2NFs and 3NFs. These are all currently generated outside JupiterNCSM. Some of these codes, for example the pAntoine code which creates the basis, are

not open source. This consideration is very important if JupiterNCSM is to be used by a larger community of researchers.

Reduce the number of data files. The external code Anicre computes transition densities for constructing the Hamiltonian matrix elements [31, 2]. This code generates a large number of data files. In the ^8Be , $N_{\text{max}} = 8$ and $N_{\text{max}} = 10$ cases this translates to hundreds of thousands of files. This is an issue, both in terms of computational performance and in terms of usability. HPC systems typically enforce file-count limitations on storage allocations, which means that these quotas are filled very quickly when using JupiterNCSM. This makes it difficult to use the software. Reducing the number of files is an important issue that should be addressed before continuing to use JupiterNCSM for larger nuclei and model spaces.

Revisit data types to reduce memory needs. Most scalars in JupiterNCSM are stored as double-precision floating point numbers. This data type is often preferred over single-precision floating point numbers when accuracy is an important factor, or to avoid issues with numerical artifacts. But a downside with using this larger data type is that it requires more storage, and consequently also longer data loading times. This is important because disk storage throughput has been identified as a major performance bottleneck. A possible improvement would be to store the matrix element data as single-precision numbers. The Krylov vectors in the Lanczos algorithm could continue to be stored as double-precision numbers. This would increase performance without sacrificing precision. It would also be possible to rewrite the entire JupiterNCSM software to use single-precision numbers instead. This, however, might cause side effects such as loss of precision or numerical instabilities in the Lanczos algorithm. This change would then be a relatively large undertaking, especially considering that validation runs would need to be run again, to guarantee that the program performs as expected. If it is possible to use single-precision numbers then it might also be possible to use half-precision numbers for some data, further decreasing storage space requirements by a factor of two.

Bibliography

- [1] Bruce R. Barrett, Petr Navratil, and James P. Vary. “Ab initio no core shell model”. In: *Prog. Part. Nucl. Phys.* 69 (2013), pp. 131–181. DOI: 10.1016/j.pnpnp.2012.10.003.
- [2] Tor Djärv. “JupiterNCSM: A Pantheon of Nuclear Physics. -an implementation of three-nucleon forces in the no-core shell model”. PhD thesis. Department of Physics, Chalmers University of Technology, Gothenburg, Sweden, 2021. ISBN: 978-91-7905-552-3.
- [3] R. Machleidt and D. R. Entem. “Chiral effective field theory and nuclear forces”. In: *Phys. Rept.* 503 (2011), pp. 1–75. DOI: 10.1016/j.physrep.2011.02.001. arXiv: 1105.2919 [nucl-th].
- [4] Evgeny Epelbaum, Hans-Werner Hammer, and Ulf-G. Meissner. “Modern Theory of Nuclear Forces”. In: *Rev. Mod. Phys.* 81 (2009), pp. 1773–1825. DOI: 10.1103/RevModPhys.81.1773. arXiv: 0811.1338 [nucl-th].
- [5] Riccardo Penco. “An Introduction to Effective Field Theories”. In: (June 2020). arXiv: 2006.16285 [hep-th].
- [6] T. Djärv et al. “Bayesian predictions for A=6 nuclei using eigenvector continuation emulators”. In: *Phys. Rev. C* 105.1 (2022), p. 014005. DOI: 10.1103/PhysRevC.105.014005. arXiv: 2108.13313 [nucl-th].
- [7] S. Wesolowski et al. “Rigorous constraints on three-nucleon forces in chiral effective field theory from fast and accurate calculations of few-body observables”. In: *Phys. Rev. C* 104.6 (2021), p. 064001. DOI: 10.1103/PhysRevC.104.064001. arXiv: 2104.04441 [nucl-th].
- [8] Isaiah Shavitt and Rodney J. Bartlett. *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*. Cambridge Molecular Science. Cambridge University Press, 2009. DOI: 10.1017/CB09780511596834.
- [9] Tor Djärv. “Three-nucleon forces in nuclear physics simulations”. Licentiate thesis. Department of Physics, Chalmers University of Technology, Gothenburg, Sweden, 2019, p. 2. URL: <https://research.chalmers.se/en/publication/512770> (visited on 2021-06-06).
- [10] J.J Sakurai and Jim Napolitano. *Modern Quantum Mechanics*. second edition. Cambridge, United Kingdom: Cambridge University Press, 2017, pp. 18, 20.
- [11] Cornelius Lanczos. “An iteration method for the solution of the eigenvalue problem of linear differential and integral operators”. In: *J. Res. Natl. Bur. Stand. B* 45 (1950), pp. 255–282. DOI: 10.6028/jres.045.026.
- [12] B. Alex Brown. “Lecture Notes in Nuclear Structure Physics”. National Superconducting Cyclotron Laboratory and Department of Physics and Astronomy. Michigan State University, E. Lansing, MI 48824, Nov. 2005.

- [13] Philip Sternberg et al. “Accelerating configuration interaction calculations for nuclear structure”. In: *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 2008, pp. 1–12. DOI: 10.1109/SC.2008.5220090.
- [14] C. Forssén et al. “Large-scale exact diagonalizations reveal low-momentum scales of nuclei”. In: *Phys. Rev. C* 97.3 (2018), p. 034328. DOI: 10.1103/PhysRevC.97.034328. arXiv: 1712.09951 [nucl-th].
- [15] *Tetralith*. National Supercomputer Centre, Linköping University. URL: <https://www.nsc.liu.se/systems/tetralith/> (visited on 2021-08-27).
- [16] *NSC Centre Storage*. National Supercomputer Centre, Linköping University. URL: <https://www.nsc.liu.se/storage/snic-centrestorage/> (visited on 2022-02-14).
- [17] *Current SNIC Large Storage Projects*. Swedish National Infrastructure for Computing. URL: <https://supr.snic.se/public/project/?type=SNIC%20Large%20Storage> (visited on 2022-03-02).
- [18] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. Nov. 2015. URL: <https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf> (visited on 2022-03-24).
- [19] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 3.1. June 4, 2015. URL: <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf> (visited on 2022-03-24).
- [20] *Driving Exascale Computing and HPC with Intel*. Intel. URL: <https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html> (visited on 2022-03-03).
- [21] *About InfiniBand*. InfiniBand Trade Association. URL: <https://www.infinibandta.org/about-infiniband/> (visited on 2022-03-02).
- [22] *MPICH Overview*. URL: <https://www.mpich.org/about/overview/> (visited on 2022-03-03).
- [23] *Open MPI: Open Source High Performance Computing*. 2022. URL: <https://www.openmpi.org/> (visited on 2022-03-03).
- [24] *MACH-2 Technology Paper*. Seagate. 2020, p. 3. URL: <https://www.seagate.com/files/www-content/solutions/mach-2-multi-actuator-hard-drive/files/tp714-dot-2-2006us-mach-2-technology-paper.pdf> (visited on 2022-03-03).
- [25] *Samsung V-NAND SSD 980 PRO. 2021 Data Sheet*. Revision 2.1. Samsung. 2021, p. 3. URL: https://semiconductor.samsung.com/resources/data-sheet/Samsung-NVMe-SSD-980-PRO-Data-Sheet_Rev.2.1.pdf (visited on 2022-03-03).
- [26] National Supercomputer Centre, Linköping University. URL: <https://www.nsc.liu.se/support/batch-jobs/tetralith/> (visited on 2022-02-18).
- [27] A. Ekström et al. “Accurate nuclear radii and binding energies from a chiral interaction”. In: *Phys. Rev. C* 91.5 (2015), p. 051301. DOI: 10.1103/PhysRevC.91.051301. arXiv: 1502.04682 [nucl-th].
- [28] A. Ekström et al. “Optimized Chiral Nucleon-Nucleon Interaction at Next-to-Next-to-Leading Order”. In: *Phys. Rev. Lett.* 110.19 (2013), p. 192502. DOI: 10.1103/PhysRevLett.110.192502. arXiv: 1303.4674 [nucl-th].

- [29] TUNL Nuclear Data Evaluation Project. *Energy Level Diagram, ^8Be* . 1988. URL: https://nuclldata.tunl.duke.edu/nuclldata/figures/08figs/08_04_1988.pdf (visited on 2022-04-28).
- [30] T. Djärv. Private correspondence. 2022.
- [31] Daniel Sääf. “Bridging scales in nuclear physics. Microscopic description of clusterization in light nuclei”. PhD thesis. Göteborg, Sweden: Department of Physics, Chalmers University of Technology, 2016.

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022
www.chalmers.se



CHALMERS
UNIVERSITY OF TECHNOLOGY