Master in High Performance Computing

Large-Scale Implementation of the Density Matrix Renormalization Group Algorithm

Supervisors:
Marcello Dalmonte,
Ivan Girotto

Candidate:
James Vance

3rd edition
2016–2017

Abstract

The density matrix renormalization group (DRMG) algorithm, a numerical technique that has been successfully used for investigating the low energy properties of one-dimensional (1D) strongly correlated quantum systems, has recently emerged as an effective tool for studying two-dimensional (2D) systems as well. At the core of DMRG is a general decimation procedure that allows the systematic truncation of the Hilbert space leaving only the most relevant basis states. However, studying 2D systems requires more degrees of freedom and greater computational resources. To address this computational roadblock, we develop a massively parallel implementation of the DMRG algorithm that targets a large number of basis states. It relies on parallel linear algebra libraries that distribute the generation and diagonalization of large sparse matrices, as these remain to be the most time-consuming steps in DMRG. We tailor our developed code for efficient performance on two sections of CINECA Marconi, a class Tier-0 supercomputing infrastructure, and evaluate its performance and scalability on up to thousands of processors. From the performance analysis we identify some limitations in scalability and suggest possible ways to rectify them.

Acknowledgments

I would like to express my deepest gratitude to my supervisors, Marcello Dalmonte and Ivan Girotto, for their expertise, guidance, and support from the initial stages up to the writing of this thesis. I would also like to thank Marlon Brenes Navarro for sharing his knowledge on PETSc and SLEPc, and Noel Lamsen for the helpful discussions on the DMRG algorithm.

I would like to acknowledge the ICTP Programme for Training and Research in Italian Laboratories (TRIL) for funding my participation in the MHPC programme. I would also like to thank CINECA and ICTP for providing access to a world-class supercomputing infrastructure.

Contents

CHAPTER 1

Introduction

The study of two-dimensional many-body quantum systems is crucial for the understanding of fundamental and technology-relevant problems such as high-temperature superconductivity [1], frustrated magnetism [2, 3], and topological phases of matter [4].

From the physics side, the study of two-dimensional (2D) systems is challenging - quantum fluctuations are too strong to reliably apply mean field methods that work in three-dimensional systems. Also, the advanced theoretical machinery which works in one-dimensional (1D) systems does not work here. Thus, the study of 2D systems often calls for computational approaches [2].

The density matrix renormalization group (DMRG), a method that was originally formulated for solving the ground state properties of 1D quantum spin systems [5], has recently been proven effective for 2D systems as well [6–8]. It has especially shown to be very useful in problems that are intractable with traditional methods such as exact diagonalization (due to large number of basis states) [9], and quantum Monte Carlo (due to the negative sign problem) [10].

DMRG is a well-understood, numerically stable and widely-applicable algorithm. The accuracy of its results can be systematically quantified by changing two control parameters - the number of states m and number of sweeps N_s [11]. However, even for 1D systems, DMRG can become computationally heavy. In this study, we will implement DMRG calculations for distributed memory architectures and enable the developed code to perform efficiently on a Tier-0 world class HPC infrastructure such as CINECA Marconi. Ultimately, our goal is to perform parallel large-scale DMRG on 1D and 2D frustrated magnets. Solving this HPC problem will give further insight into unprecedented regimes of physical problems.

CHAPTER 2

Background

2.1 Strongly Correlated Quantum Systems

The many-body problem is one of the grand challenges in quantum many-body systems. This is due to the large number of degrees of freedom needed to represent a quantum state. Oftentimes, this number grows exponentially with the size N of the system in consideration [12]. Physical systems can be effectively simplified into single-particle models when many-particle interactions are treated perturbatively. However, this simplification breaks down in the case of strongly-correlated systems in which the interactions among constituents of the system are non-trivial.

In recent years, several methods have been developed to study strongly-correlated quantum systems. A common theme among these methods is that they are most effective in finding the ground state energy wavefunction of the system from which one can calculate observables. These standard methods include exact diagonalization, quantum Monte Carlo, and the density matrix renormalization group algorithm.

Within *exact diagonalization* (ED), the Hamiltonian of a quantum system is represented by a sparse matrix whose eigenvectors and eigenvalues may be obtained using several direct and iterative methods such as Krylov subspace techniques. However, even with many simplifying assumptions and symmetries, ED still suffers from exponential growth of the required number of basis states. Consequently, the computational resources needed to calculate even a portion of its eigenspectrum also grows exponentially with the system size [13].

Quantum Monte Carlo (QMC) algorithms use importance sampling to solve higher-order integrals in the calculation of expectation values, and thus are able to bypass this

increased complexity. Although the number of possible configurations still scale exponentially with the number of particles, with QMC, the computational effort now scales as a polynomial in N [14]. QMC methods have been used successfully for bosonic systems. However, these methods often fail in fermionic and frustrated spin systems due to the so called 'sign problem', that invalidates importance sampling, resulting in exponential increase in time to solution with the system size [10].

Addressing the limitations of ED and QMC is the *Density Matrix Renormalization Group* (DMRG) algorithm, which has recently emerged as a numerically precise and highly reliable method for studying many-body quantum systems [5]. Its efficient decimation procedure allows for the study of large systems intractable by ED, and it does not suffer from the frustrated spin or fermionic sign problem found in QMC [15].

2.2 The DMRG Algorithm

The Density Matrix Renormalization Group (DMRG) algorithm is a variational method used to study the ground state properties of low-dimensional many-body quantum systems such as spin lattices or molecular orbitals. The kernel of this method is to obtain a truncated wavefunction in a reduced Hilbert space whose basis is chosen to minimize the loss of information. This is achieved by decomposing the lattice into smaller sub-blocks and iteratively increasing the size of each block while performing a truncation of the operator bases at each step. With DMRG, the error resulting from truncation can be measured and controlled, and large systems can be treated very accurately to obtain ground state energies and gaps [11].

The method was conceptualized in 1992 by Steven White as an improvement to the block decimation procedure in Wilson's Numerical Renormalization Group (NRG) [5, 16]. Since then, DRMG has evolved and has been applied to different problems in domains outside of condensed matter including nuclear physics [17, 18] and quantum chemistry [19, 20].

In the following section we illustrate the traditional DMRG algorithm following the original paper [5] and subsequent detailed reviews and tutorials [11, 15, 21]. For simplicity we will be using spins in our description. However, the same procedure is immediately applicable to fermions and bosons.

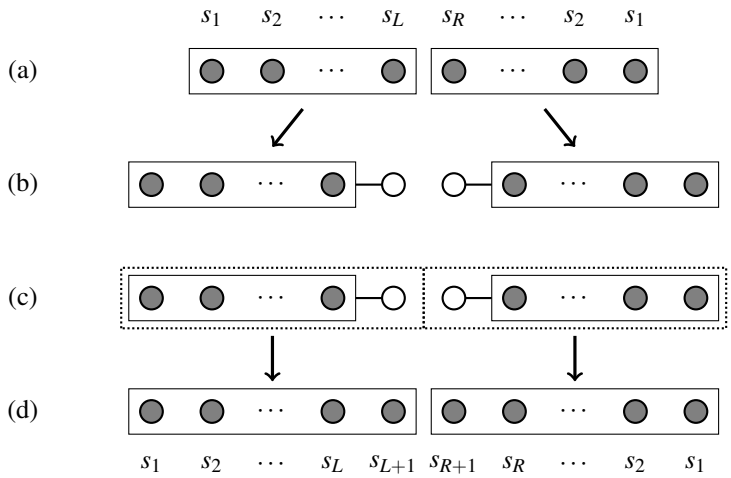

Figure 2.1: Block growing scheme in the infinite-size DMRG algorithm: Starting with the blocks of length L (R) and basis size D_L (D_R) formed in the previous step (a), we grow each block by adding one site at their interface (b). We construct the superblock Hamiltonian and diagonalize it to obtain the ground state (c). We calculate the reduced density matrix for each block and use its m largest eigenvectors to rotate the operators to a truncated basis for the next step (d). [15]

2.2.1 Infinite-Size Algorithm

The starting point of the algorithm is to have two blocks of spin sites, referred to as the left and right blocks, or the system and environment blocks. The traditional DMRG algorithm then proceeds in two distinct phases: the infinite-size algorithm and the finite-size algorithm. The infinite-size DMRG (iDMRG) algorithm serves as the warm-up stage for the full DMRG algorithm since the resulting block and site operators may be stored to perform the finite-size algorithm. Its implementation also provides a template for the finite-size algorithm as the same steps will be performed in this procedure.

The algorithm, illustrated in Figure 2.1, starts with the left (L) and right (R) blocks each containing one site with local dimension d. These blocks are denoted by $\mathfrak{B}(1,d)$ where the arguments indicate the length and dimension of the block, respectively. We construct all the operator matrices, $\hat{O}$, required to represent the interaction between sites. For simplicity in our illustration of the algorithm, we consider systems with nearest-neighbor interactions in which the Hamiltonian usually takes the form

$$\mathbf{H} = \sum_i \sum_n (a_n \mathbf{S}_{n,i} \mathbf{T}_{n,i+1} + b_n \mathbf{V}_{n,i}) \tag{2.1}$$

where a_n and b_n are coupling constants, and $\mathbf{S}_{n,i}$, $\mathbf{T}_{n,i}$ and $\mathbf{V}_{n,i}$ are operators acting on

the i^{th} site, and n serves as index for the various terms of the Hamiltonian. [21]

From a block of length L, the resulting enlarged block with $L+1$ sites has a Hamiltonian matrix of the form

$$\mathbf{H}_{L\circ} = \mathbf{H}_L \otimes \mathbf{1}_\circ + \mathbf{1}_L \otimes \mathbf{H}_\circ + \mathbf{H}_{L\leftrightarrow\circ} \tag{2.2}$$

and correspondingly, for the right block with R sites enlarged to $R+1$ the Hamiltonian is given by

$$\mathbf{H}_{\circ R} = \mathbf{1}_\circ \otimes \mathbf{H}_R + \mathbf{H}_\circ \otimes \mathbf{1}_R + \mathbf{H}_{\circ\leftrightarrow R} \tag{2.3}$$

where $\otimes$ is the Kronecker product, the circle symbol $\circ$ denote the added site, the last terms indicate the interaction between the block and the added sites, and $\mathbf{1}$ is the identity matrix corresponding to the block or site subspace denoted as its index.

Initially, we perform the addition of sites in a numerically exact manner until the blocks grow to $\mathfrak{B}(n, D_{L(R)})$ where the number of states $D_{L(R)} = d^n$ needed to exactly represent the block is just below or equal to our desired maximum number of basis states m. After reaching this threshold, the blocks are further enlarged but a truncation procedure is performed at each successive iteration that ensures that we keep a maximum number of states to describe the enlarged block. Taking the interactions of the identical enlarged blocks $\mathfrak{B}(n+1, dD_{L(R)})$, we form the superblock Hamiltonian

$$\mathbf{H}_{L\circ\circ R} = \mathbf{H}_{L\circ} \otimes \mathbf{1}_{\circ R} + \mathbf{1}_{L\circ} \otimes \mathbf{H}_{\circ R} + \mathbf{H}_{L\circ\leftrightarrow\circ R}. \tag{2.4}$$

where the last term denotes the interaction between the connecting sites of the two blocks. [11, 21]

We solve the ground state of the superblock Hamiltonian corresponding to the eigenvalue problem posed by the time-independent Schrödinger equation

$$\mathbf{H}_{L\circ\circ R}\,|\psi_{GS}\rangle = E_{GS}\,|\psi_{GS}\rangle \tag{2.5}$$

using iterative diagonalization techniques such as Lanczos or Davidson algorithms. The ground state vector takes the form

$$|\psi_{GS}\rangle = \sum_{l,r} \psi_{l,r}\,|l\rangle \otimes |r\rangle \tag{2.6}$$

expressed in terms of the basis vectors of the left and right enlarged blocks. From the ground state eigenvector, we calculate the reduced density matrices of each block by tracing out the other block

$$\rho_{L\circ} = \mathrm{Tr}_{\circ R} \left|\psi_{GS}\right\rangle \left\langle\psi_{GS}\right| \tag{2.7}$$

$$\rho_{\circ R} = \mathrm{Tr}_{L\circ} \left|\psi_{GS}\right\rangle \left\langle\psi_{GS}\right| \tag{2.8}$$

We take the full eigenvalue decomposition of each block's density matrix and use this to express the reduced density matrices as

$$\rho_{L\circ} = \sum_{\alpha} \omega_{\alpha} \left|\alpha\right\rangle_{L\circ\, L\circ} \langle\alpha|. \tag{2.9}$$

Taking the vectors with m largest weights ω_{α}, we construct a rotation matrix $U_{L\circ}$ of dimension $dD_L \times m$ where $D_L = m$ if a previous truncation has been performed. We use this to rotate the operator matrices $O_{L\circ}$ of dimension dD_L acting on the enlarged block to

$$\tilde{\mathbf{O}}_{L\circ} = (U_{L\circ})^{\dagger}\, \mathbf{O}_{L\circ} U_{L\circ} \tag{2.10}$$

The transformed operator matrix now has dimension $\min(m, dD_L)$. We also perform the corresponding steps on the enlarged right block $\circ R$. After setting $L \leftarrow L\circ$ and $R \leftarrow \circ R$ as the starting blocks, the block enlargement and basis truncation steps are repeated. The density matrix truncation is applied on both blocks at each successive iteration until a suitable system size or tolerance in the error of the energy is reached. [11, 15]

2.2.2 Application to the Heisenberg Spin Chain

The Heisenberg model is one of the simplest but non-trivial models studied with DMRG. Given a one-dimensional spin-$1/2$ chain of length N as illustrated in Figure 2.2, the Heisenberg Hamiltonian is defined as

$$\mathbf{H} = \sum_{i=1}^{N-1} J_i \mathbf{S}_i \mathbf{S}_{i+1} \tag{2.11}$$

where J_i is a coupling constant, and $\mathbf{S} = (\mathbf{S}^x, \mathbf{S}^y, \mathbf{S}^z)$ are the spin operators represented as Pauli matrices for spin-$1/2$ chains. Introducing the raising and lowering operators

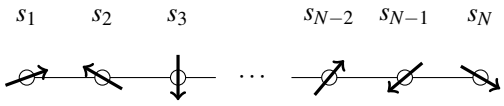

Figure 2.2: One-dimensional spin chain with open boundary conditions and nearest-neighbor interaction.

$\mathbf{S}^{\pm} = \mathbf{S}^x \pm i\mathbf{S}^y$ allows us to rewrite the Hamiltonian into the more convenient form

$$\mathbf{H} = \sum_{i=1}^{N-1} \left(J_z \mathbf{S}_i^z \mathbf{S}_{i+1}^z + \frac{J}{2} \left[\mathbf{S}_i^+ \mathbf{S}_{i+1}^- + \mathbf{S}_i^- \mathbf{S}_{i+1}^+ \right] \right) \tag{2.12}$$

where J and J_z are coupling constants and the index i runs over interactions between nearest-neighbor sites. For spin-$1/2$ systems with $J = J_z = 1$, the Bethe ansatz was used to exactly solve for the ground state energy which was found to be $E_0 = 1/4 - \ln(2) \approx 0.4431471805599$. Several DMRG studies have also explored the low-energy properties of the Heisenberg model in different geometries and higher dimensions such as two-dimensional ladders [8] and the Kagome lattice [22]. For our implementation, we will be using the spin-$1/2$ one-dimensional Heisenberg model as test Hamiltonian due to its simplicity and the presence of an exact solution for comparison.

A single site of a spin-$1/2$ chain is described by a state with the basis vectors $|\uparrow\rangle$ and $|\downarrow\rangle$ forming a local dimension of $d = 2$. To explicitly construct (2.12), we use the following matrix representations of the single-site spin operators

$$\mathbf{S}_\circ^z = \begin{pmatrix} 1/2 & 0 \\ 0 & -1/2 \end{pmatrix}, \quad \mathbf{S}_\circ^+ = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad \mathbf{S}_\circ^- = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \tag{2.13}$$

which are derived from the Pauli matrices. With respect to the entire N-spin chain, an operator acting on the i^{th} site is explicitly represented by

$$\mathbf{S}_{i,N}^q = \left(\bigotimes_{j=1}^{i-1} \mathbf{1}_\circ \right) \otimes \mathbf{S}_\circ^q \otimes \left(\bigotimes_{j'=i+1}^{N} \mathbf{1}_\circ \right) \tag{2.14}$$

where $q = \{z, +, -\}$ and $\mathbf{1}_\circ$ is the $d \times d$ identity matrix. If this acts on the last site of the left (right) block of length L (R), we simply denote it as $\mathbf{S}_{L(R)}^q$, and on the last site of the enlarged block of length $L+1$ ($R+1$) as $\mathbf{S}_{L\circ(\circ R)}^q$. With this expression, we can explicitly construct the Hamiltonian (2.12) and the spin operators at each site.

Alternatively, we can also construct the Hamiltonian iteratively using (2.2) and (2.12). We begin with the case of two spins and construct its Hamiltonian explicitly as

$$\mathbf{H}_2 = J_z \mathbf{S}^z_\circ \otimes \mathbf{S}^z_\circ + \frac{J}{2}\left[\mathbf{S}^+_\circ \otimes \mathbf{S}^-_\circ + \mathbf{S}^-_\circ \otimes \mathbf{S}^+_\circ\right]. \tag{2.15}$$

Then we can iteratively enlarge the block by adding another site to each block so that the Hamiltonian of a left (right) enlarged block containing $L+1 (R+1)$ sites is expressed by the recursion

$$\mathbf{H}_{L\circ} = \mathbf{H}_L \otimes \mathbf{1}_\circ + J_z \mathbf{S}^z_L \otimes \mathbf{S}^z_\circ + \frac{J}{2}\left[\mathbf{S}^+_L \otimes \mathbf{S}^-_\circ + \mathbf{S}^-_L \otimes \mathbf{S}^+_\circ\right] \tag{2.16}$$

$$\mathbf{H}_{\circ R} = \mathbf{1}_\circ \otimes \mathbf{H}_R + J_z \mathbf{S}^z_\circ \otimes \mathbf{S}^z_R + \frac{J}{2}\left[\mathbf{S}^+_\circ \otimes \mathbf{S}^-_R + \mathbf{S}^-_\circ \otimes \mathbf{S}^+_R\right]. \tag{2.17}$$

where $\mathbf{S}^q_L = \mathbf{S}^q_{L,L}$ and $\mathbf{S}^q_R = \mathbf{S}^q_{R,R}$ are the operators of the last sites added to the left and right blocks with (possibly truncated) basis dimensions D_L and D_R, respectively. Using (2.4) we can now construct the superblock Hamiltonian as

$$\begin{aligned}\mathbf{H}_{L\circ\circ R} = {} & \mathbf{H}_{L\circ} \otimes \mathbf{1}_{\circ R} + \mathbf{1}_{L\circ} \otimes \mathbf{H}_{\circ R} \\ & + J_z \mathbf{S}^z_{L\circ} \otimes \mathbf{S}^z_{\circ R} + \frac{J}{2}\left[\mathbf{S}^+_{L\circ} \otimes \mathbf{S}^-_{\circ R} + \mathbf{S}^-_{L\circ} \otimes \mathbf{S}^+_{\circ R}\right]\end{aligned} \tag{2.18}$$

with a total length $L+R+2$ and a basis size of $d^2 D_L D_R$. [11]

2.2.3 Quantum Numbers and Symmetries

For many systems studied with DMRG, the Hamiltonian conserves some quantum numbers such that it commutes with the corresponding operator. These symmetries may be exploited to decrease the Hilbert space dimension which reduces memory consumption and computation time. Here, we discuss the most frequently used symmetry, the $U(1)$ symmetry, in which total magnetization and/or total particle number are conserved quantum numbers. In the context of the Heisenberg model, $U(1)$ symmetry conserves the total magnetization $S^z_{\text{tot}} = \sum_{i=1}^N S^z_j$ allowing us to express the operators in terms of smaller blocks corresponding to a set of quantum numbers. [15]

In our particular spin-$1/2$ model, S^z_j are the eigenvalues of the $\mathbf{S}^z$ operator (2.13) at site j which are simply $\left\{+\frac{1}{2}, -\frac{1}{2}\right\}$. Thus, for a block of N sites, the possible quantum numbers are $\left\{\frac{N}{2}, \frac{N}{2}-1, \ldots, -\frac{N}{2}\right\}$. If we label the block and site states of the subsystems with their magnetizations and and preserve the total magnetization S^z_{tot} of the

superblock, we get

$$S^z_{\text{tot}} = S^z_L + S^z_{\sigma^L} + S^z_{\sigma^R} + S^z_R \tag{2.19}$$

where $\sigma^{L(R)}$ denote the added sites to the left and right blocks respectively. Thus, we can target a particular magnetization sector for the ground state solution (2.5), usually at $S^z_{\text{tot}} = 0$, or study spin gaps by solving the ground state at different target magnetizations. The reduced density matrices (2.8) also acquire a block-diagonal structure which allows us to perform its spectral decomposition (2.9) on each of these diagonal blocks. [15]

2.3 Libraries for Massively Parallel Linear Algebra

While simple DMRG applications require only moderate resources, going towards higher dimensions is computationally challenging since it requires keeping a large number of states. This results in large matrices and computationally intensive operations. Thus there is a need for a parallel implementation that uses scalable linear algebra libraries for the management of data and communication, and the implementation of the various solvers.

The *Portable, Extensible Toolkit for Scientific Computation* (PETSc) is a suite of data structures and routines for the scalable parallel solution of scientific applications originally designed for solving partial differential equations. It uses Message Passing Interface for distributed memory parallelism. It includes an implementation of parallel matrices and vectors together with basic operations on these objects, and several linear system solvers. [23] One of the packages that extends upon PETSc is the *Scalable Library for Eigenvalue Problem Computations* (SLEPc) which is used for the eigenvalue decomposition of large sparse matrices on parallel distributed-memory architectures. Together, these two packages provide the framework in which we implement our scalable DMRG calculations. [24] In this section we will explain the general features of these packages and what makes them suitable for our DMRG calculations.

PETSc implements a class of sparse matrices called `MATAIJ` which stores the elements based in a compressed sparse row (CSR) format. In the CSR format the non-zero elements of each row are represented as a list of integer column indices and floating-point values. Distributed memory storage of this matrix is achieved by assigning ownership of an almost uniform number of rows to each MPI process. Correspondingly, vectors are also stored in a distributed manner following the layout of the matrix which

acts on it.

For ubiquitous matrix-vector product operations, overlap between communication and computation is achieved by separately storing a diagonal block of matrix elements that act on the local portion of the vector, and an off-diagonal block of matrix elements that act on the non-local elements of the vector that are gathered to complete the operation [25]. Thus, when constructing a matrix operator, knowing how much memory to preallocate for the diagonal and off-diagonal blocks is essential for improved performance. PETSc also has a `MATSHELL` matrix-type with a customizable interface that allows for a matrix-free approach especially for matrix-vector products. [23]

On the other hand, SLEPc contains eigensolvers that are most efficient for finding a small number eigenvalues of large sparse matrices based on several iterative algorithms. It already includes the usual methods that are used for Hamiltonian matrices such as Lanczos, Arnoldi and Krylov-Schur algorithms. Additionally, it also provides functionality for the partial SVD of rectangular matrices, solutions to generalized non-Hermition and non-linear eigenvalue problems, and some interface to serial LAPACK routines. [24]

The combination of PETSc and SLEPc provides the basic functionalities needed for a large-scale DMRG calculation, including parallel sparse eigensolvers and basic linear algebra operations. They also enable scalability by using distributed-memory parallelism. Although the DMRG workflow is inherently serial, the generation of and operation on large sparse operator matrices can greatly benefit from the speedup brought about by this parallelism. In fact, a previous thesis has demonstrated a massively-parallel implementation for the time-evolution of disordered quantum systems using Krylov subspace techniques inherently provided by these libraries [26, 27].

CHAPTER 3

Implementation

In this chapter, we present our implementation of the infinite-size DMRG algorithm as detailed in the previous chapter. In Section 3.1 we describe the most basic but widely-applicable implementation that does not take symmetries into account but rather solves the entire Hamiltonian. We present an optimized version of this implementation in Section 3.2 by targeting a specific magnetization sector of the Hamiltonian which dramatically decreases the overall computational cost. For concreteness, we shall use the one-dimensional spin-1/2 Heisenberg chain with local dimension $d = 2$ as explicit model for our construction while keeping the notation as general as possible so that the same procedure can also be applied to other systems.

3.1 Basic Implementation

We first discuss the basic implementation which follows the traditional formulation in Section 2.2.1 where quantum numbers and symmetries are not considered. This approach is more computationally intensive but its implementation is simpler and it is applicable to a wider range of physical problems.

3.1.1 Workflow

Our implementation starts with the construction of the d-by-d operators $\mathbf{H}_\circ$, $\mathbf{S}^z_\circ$ and $\mathbf{S}^+_\circ$ for the single site. The operator $\mathbf{S}^-_\circ$ and its representation relative to a block can be obtained from the Hermitian conjugate $\mathbf{S}^-_{i,N} = (\mathbf{S}^+_{i,N})^\dagger$ whenever needed. With these matrices, we form the left and right blocks containing one site each and perform the

steps of the DMRG iteration. Based on the details provided in Section 2.2.1, we divide a single iteration of the DMRG algorithm into several distinct phases which we have implemented as functions in a C++ class called `iDMRG`.

- BuildBlockLeft and BuildBlockRight involve the addition of a site to the left and right blocks to form the enlarged blocks $L\circ$ and $\circ R$, respectively. These steps involve the calculation of the enlarged block Hamiltonians $\mathbf{H}_{L\circ}$ ($\mathbf{H}_{\circ R}$) given in (2.2)-(2.3), and the construction of the operators $\mathbf{S}^z_{L\circ}$ and $\mathbf{S}^+_{L\circ}$ ($\mathbf{S}^z_{\circ R}$ and $\mathbf{S}^+_{\circ R}$) for the newly added sites using (2.14). Using (2.14), the operators $\mathbf{S}^z_{L\circ}$ and $\mathbf{S}^+_{L\circ}$ ($\mathbf{S}^z_{\circ R}$ and $\mathbf{S}^+_{\circ R}$) for the newly added sites are also constructed relative to their respective blocks. These steps result in the creation of several matrices of size $dD_{L(R)} \times dD_{L(R)}$. [28]
- BuildSuperblock constructs the superblock Hamiltonian $\mathbf{H}_{L\circ\circ R}$ following equation (2.4) which results in a large sparse matrix of size $d^2D_LD_R \times d^2D_LD_R$ in the `MATAIJ` format.
- SolveGroundState obtains the ground state solution of the superblock Hamiltonian by solving the eigenvalue problem posed in (2.5).
- BuildReducedDMs constructs the reduced density matrices $\rho_{L\circ}$ and $\rho_{\circ R}$ from the ground state eigenvector following (2.8). This results in smaller matrices with maximum size $dD_{L(R)} \times D_{L(R)}$.
- GetRotationMatrices is where we solve for the full spectrum of the reduced density matrices using SLEPc's Singular Value Decomposition (SVD) class which has an interface to dense LAPACK routines. It also constructs the rotation matrix $U_{L\circ}$ using the singular vectors of the reduced density matrices.
- TruncateOperators performs the rotation of the block Hamiltonian and site operators using (2.10).

3.1.2 Kronecker Product of Distributed Sparse Matrices

The key operation in the construction of matrix operators for the enlarged blocks and the superblock Hamiltonian is the Kronecker product. Given two matrices $\mathbf{A}$ and $\mathbf{B}$ of dimension (M_A, N_A) and (M_B, N_B), respectively, the Kronecker product $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ of dimension (M_AM_B, N_AN_B) has elements given by

$$C_{\gamma,\delta} = A_{i,j} \cdot B_{k,l} \tag{3.1}$$

Algorithm 1 Local Row and Column Ownership of Global Matrices

Input: Global matrix size M_{global}, MPI communicator rank n_{rank} and size n_{procs}
Output: Row ownership range $[i_{\text{start}}, i_{\text{end}})$

```
function SplitOwnershipRange( M_global, n_rank, n_procs)
    i_offset ← (M_global mod n_procs)
    m_local ← M_global / n_procs                ▷ Number of locally-owned rows/columns
    if n_rank < i_offset then                   ▷ Distribute remainder to first few processes
        i_offset ← n_rank
        m_local ← m_local + 1
    end if
    i_start ← M_global / n_procs · n_rank + i_offset
    i_end ← i_start + m_local
    return i_start, i_end
end function
```

where $\gamma = i \cdot M_B + k$ and $\delta = j \cdot N_B + l$. Looking at (2.2)-(2.4), we can see that the matrices that will be constructed often involve a linear combination of Kronecker products of the form

$$\mathbf{C} = \sum_n a_n \mathbf{A}_n \otimes \mathbf{B}_n \tag{3.2}$$

where n is an index over terms and a_n is a coefficient. As of version 3.7.6, PETSc does not yet support this operation for general sparse matrices so we have written our own custom implementation with the simplifying assumption that the sets of matrices $\{\mathbf{A}_n\}$ and $\{\mathbf{B}_n\}$ will have uniform dimensions among themselves.

There are several constraints to our implementation. First, the matrix operators are stored in a compressed sparse row (CSR) format which is more efficient for accessing and writing elements contiguously in row-major order than in column-major order. Second, row ownership is distributed among all MPI processes in the communicator so that some rows of $\mathbf{A}_n$ and $\mathbf{B}_n$ must be sent to the correct processes where they are needed to construct the local rows of resultant matrix $\mathbf{C}$. Finally, preallocation for the non-zeros of the matrix must be done to improve efficiency during matrix construction so the resulting number of non-zeros of $\mathbf{C}$ must be known before construction. Taking these into account, we divide our implementation of the Kronecker product into three stages: *submatrix collection*, *preallocation*, and *matrix construction*.

In submatrix collection, we determine and communicate the entire rows of $\mathbf{A}_n$ and $\mathbf{B}_n$ needed to construct the local rows of $\mathbf{C}$ for each MPI process. We begin by calculating the dimensions of $\mathbf{C}$ and determining the row ownership range of the resultant

Algorithm 2 Kronecker Product: Submatrix Collection

Input:
MPI matrices $\{\mathbf{A}_n\}_{n=1}^{M}$ of size (M_A,N_A) and $\{\mathbf{B}_n\}_{n=1}^{M}$ of size (M_B,N_B)
MPI communicator rank n_{rank} and size n_{procs}
Output:
Sequential matrices $\{\mathbf{A}_{n,\text{sub}}\}_{n=1}^{M}$ and $\{\mathbf{B}_{n,\text{sub}}\}_{n=1}^{M}$ containing non-local rows
procedure KRONGETSUBMATRICES
$M_C \leftarrow M_A \cdot M_B$ ▷ Global number of rows of $\mathbf{C}$
$N_C \leftarrow N_A \cdot N_B$ ▷ Global number of columns of $\mathbf{C}$
$i_{C,\text{start}}, i_{C,\text{end}} \leftarrow$ SPLITOWNERSHIPRANGE(M_C, n_{rank}, n_{procs})
for $i_C \in [i_{C,\text{start}}, i_{C,\text{end}})$ **do**
$\text{IS}_A \leftarrow \text{insert}(i_C / M_B)$ ▷ Index set of required rows of $\mathbf{A}$
$\text{IS}_B \leftarrow \text{insert}(i_C \bmod M_B)$ ▷ Index set of required rows of $\mathbf{B}$
end for
for $n \in \{1 \dots M\}$ **do**
$\mathbf{A}_{n,\text{sub}} \leftarrow$ `MatGetSubmatrices` of all rows of $\mathbf{A}_n$ listed in IS_A
$\mathbf{B}_{n,\text{sub}} \leftarrow$ `MatGetSubmatrices` of all rows of $\mathbf{B}_n$ listed in IS_B
end for
end procedure

matrix that conforms to PETSc's default layout. This is achieved by using the function SPLITOWNERSHIPRANGE shown in Algorithm 1 which outputs the range of row indices $[i_{C,\text{start}}, i_{C,\text{end}})$ belonging to the current process. In Algorithm 2 we describe the remaining procedure which involves determining the indices of the rows of $\mathbf{A}_n$ and $\mathbf{B}_n$ needed by a particular MPI process. These rows are then collected into submatrices by using PETSc's `MatGetSubmatrices` function [23]. Additionally, a mapping object is created that takes the non-local row index and maps it to the local row index in the sequential submatrix.

A compressed sparse row matrix $\mathbf{A}$ may be logically thought of as an array of rows and that each row $\mathbf{A}[i_A]$ is a list of tuples $(\text{col}_A, \text{val}_B)$ representing the column indices and values of only the non-zero elements in each row. Knowing how many non-zeros there will be in the matrix, and preallocating the corresponding amount of memory is crucial for attaining good performance as it avoids the repeated reallocation of memory for the non-zero rows during matrix construction. During prealloaction, we determine the required storage for each row of the resultant MPI or global sparse matrix following PETSc's `MATMPIAIJ` format which is assembled into two sequential blocks in the `MATSEQAIJ` format: the *diagonal block* and the *off-diagonal block*. An element that has a column index in the range $[j_{C,\text{start}}, j_{C,\text{end}})$ calculated from Algorithm 1 belongs to the diagonal block. Otherwise, the element belongs to the off-diagonal block. The

Algorithm 3 Kronecker Product: Preallocation

Input: Sets of sequential submatrices $\{\mathbf{A}_{q,\text{sub}}\}_{q=1}^{M}$ and $\{\mathbf{B}_{q,\text{sub}}\}_{q=1}^{M}$
Output: Arrays DNNZ[] and ONNZ[] containing the number of non-zeros in the diagonal and off-diagonal blocks, respectively

```
procedure KRONPREALLOCATION
    j_C,start, j_C,end ← SPLITOWNERSHIPRANGE(N_C, n_rank, n_procs)
    for i_C ∈ [i_C,start, i_C,end) do
        DNNZ[i_C] ← 0, ONNZ[i_C] ← 0
        for q ∈ {1...Q} do
            for (col_A, val_A) ∈ A_q,sub[i_C/M_B] do
                for (col_B, val_B) ∈ B_q,sub[i_C mod M_B] do
                    col_C ← col_A · N_B + col_B
                    if j_C,start ≤ col_C < j_C,end then
                        DNNZ[i_C] ← DNNZ[i_C] + 1
                    else
                        ONNZ[i_C] ← ONNZ[i_C] + 1
                    end if
                end for
            end for
        end for
    end for
end procedure
```

number of elements in these blocks for each row are calculated and stored in arrays DNNZ and ONNZ for the diagonal and off-diagonal blocks, respectively, as illustrated in Algorithm 3. These arrays are then fed into the `MatMPIAIJSetPreallocation` and `MatSeqAIJSetPreallocation` functions of PETSc. [23, 29]

Finally, we use the column indices and values of the sequential submatrices to construct the local rows of $\mathbf{C}$ in the procedure KRONMATRIXCONSTRUCTION as illustrated in Algorithm 4. Here, we loop through the rows of $\mathbf{A}_n$ and $\mathbf{B}_n$, and follow the definition of the Kronecker product in (3.1). Additionally, we loop through the terms of the linear combination in (3.2). The insertion of values to the matrix is achieved using the `MatSetValues` function provided by PETSc.

Together, these procedures allow us to explicitly construct the matrix representation of the block operators and the superblock Hamiltonian during the `BuildBlockLeft`, `BuildBlockRight` and `BuildSuperblock` stages of the workflow, and distribute them among different MPI processes.

Algorithm 4 Kronecker Product: Matrix Construction

Input: Coefficients $\{a_n\}$ of the linear combination of $\otimes$
Sequential submatrices $\{\mathbf{A}_{n,\text{sub}}\}_{n=1}^{M}$ and $\{\mathbf{B}_{n,\text{sub}}\}_{n=1}^{M}$ containing non-local rows
Output: Resultant MPI matrix $\mathbf{C}$ (3.2)

```
procedure KronMatrixConstruction
    for row_C ∈ [i_C,start, i_C,end) do
        row_A ← i_C / M_B
        row_B ← i_C mod M_B
        for q ∈ {1 ... Q} do
            for (col_A, val_A) ∈ A_n,sub[row_A] do
                for (col_B, val_B) ∈ B_n,sub[row_B] do
                    col_C ← col_A · N_B + col_B
                    val_C ← a_n · val_A · val_B
                    C[row_C] ← insert((col_C, val_C))
                end for
            end for
        end for
    end for
end procedure
```

3.1.3 Ground State Solution

After building the superblock Hamiltonian, the next step is finding the ground state solution. This is represented by the time-independent Schrödinger equation $\mathbf{H}|\psi\rangle = E|\psi\rangle$ for which we solve for the eigenpair corresponding to the lowest eigenvalue. Since the Hamiltonian matrix for the system is sparse and very large, and only an extremal part of the eigenspectrum is needed, a full eigendecomposition of the matrix is impractical. Instead, iterative diagonalization techniques are often used which include the Lanczos and Davidson algorithms. These and many other techniques are implemented in SLEPc and have been tested for scalability up to thousands of cores. [24]

In particular, the Krylov-Schur algorithm, which is based on Krylov subspace methods, has shown good performance results for both Hermitian and non-Hermitian eigenproblems making it SLEPc's default eigensolver. In our case in which the Hamiltonian matrix is Hermitian, the Krylov-Schur algorithm is equivalent to a thick-restart Lanczos method which maintains an upper bound in the number of Lanczos vectors. [30, 31]

Once the Hamiltonian matrix has been constructed following the previous section, using SLEPc's Eigenvalue Problem Solver (EPS) object to find the ground state is very straightforward. This step is implemented in the `SolveGroundState` section of our

Listing 3.1 Code snippet for calling the eigensolver

```
EPS eps;
EPSCreate(PETSC_COMM_WORLD, &eps);
EPSSetOperators(eps, superblock_H_, NULL);
EPSSetProblemType(eps, EPS_HEP); // Hermitian
EPSSetWhichEigenpairs(eps, EPS_SMALLEST_REAL); // Ground state
EPSSetFromOptions(eps); // Specify options at run time
EPSSolve(eps);
```

workflow. The main portion of the calling sequence is shown in Listing 3.1.

Parameters such as which algorithm to use, the kind and number of eigenpairs to solve for, and the tolerance can be specified as run-time parameters [32]. For our purpose, we use the Krylov-Schur algorithm and we specified lowest real eigenvalues with relative tolerance set at 10^{-12}.

3.1.4 Construction of the Reduced Density Matrices

When the eigensolver has reached the desired tolerance, the ground state eigenvector may now be extracted from the EPS object. This vector takes the form

$$|\psi_{GS}\rangle = \sum_{l,r} \psi_{l,r} |l\rangle \otimes |r\rangle \tag{3.3}$$

where indices l and r run over the bases of the enlarged blocks $L\circ$ and $\circ R$, respectively, and it has a global length of $d^2 D_L D_R$. Its corresponding reduced density matrix takes the form

$$\rho_{L\circ\circ R} = |\psi_{GS}\rangle \langle\psi_{GS}| = \sum_{l,r,l',r'} \psi_{l,r}\psi^*_{l',r'} |l\rangle \langle l'| \otimes |r\rangle \langle r'| \tag{3.4}$$

To get the reduced density matrix for the left (right) block, we trace out the right (left) block subsystem

$$\rho_{L\circ} = \mathrm{Tr}_{\circ R} |\psi_{GS}\rangle \langle\psi_{GS}| = \sum_{l,l'} \left(\sum_{r} \psi_{l,r}\psi^*_{l',r} \right) |l\rangle \langle l'| \tag{3.5}$$

$$\rho_{\circ R} = \mathrm{Tr}_{L\circ} |\psi_{GS}\rangle \langle\psi_{GS}| = \sum_{r,r'} \left(\sum_{l} \psi_{l,r}\psi^*_{l,r'} \right) |r\rangle \langle r'| . \tag{3.6}$$

where the inner summations resemble matrix multiplication [33]. This means that we can reshape the vector $|\psi_{GS}\rangle$ in row-major order into a matrix Ψ of size $dD_L \times dD_R$ reflecting the Kronecker product structure. Then, the reduced density matrices are simply

$$\rho_{L\circ} = \Psi\Psi^\dagger \tag{3.7}$$

$$\rho_{\circ R} = \Psi^\dagger\Psi \tag{3.8}$$

where † is the conjugate transpose.

We note that the ground state vector is in a parallel layout following the row-wise distribution of the superblock Hamiltonian. Thus, to construct (3.7) and (3.8), whether in parallel or in serial, the reshaped ground state vector has to be redundantly present in the participating processors. This is not a problem since it takes up much less memory compared to the superblock Hamiltonian and it can be achieved by using the `VecScatter` functionality in PETSc. This routine is implemented in the procedure `BuildReducedDensityMatrices` detailed in Algorithm 5, with run-time options to construct the matrices in serial on process 0 or in parallel.

Algorithm 5 Construction of the reduced density matrices

Input: Global ground state vector ψ_{GS} of length $dD_L \times dD_R$
Run-time boolean option on whether to produce global output GLOBAL
Output: Reduced density matrices $\rho_{L\circ}$ ($dD_L \times dD_L$) and $\rho_{\circ R}$ ($dD_R \times dD_R$)

```
procedure BUILDREDUCEDDENSITYMATRICES
    ψ ← full local copy of ψ_GS using VecScatter
    Prepare dD_L × dD_R local matrix Ψ
    for i ∈ [0, dD_L) do
        for j ∈ [0, dD_R) do
            Ψ[i, j] ← ψ[i · dD_R + j]          ▷ Copy values in row-major order
        end for
    end for
    if GLOBAL then
        i_C,start, i_C,end ← SPLITOWNERSHIPRANGE(dD_L, n_rank, n_procs)
        ρ_L∘[rows ∈ [i_C,start ... i_C,end)] ← Ψ[rows ∈ [i_C,start ... i_C,end)] · Ψ†
        i_C,start, i_C,end ← SPLITOWNERSHIPRANGE(dD_R, n_rank, n_procs)
        ρ_∘R[rows ∈ [i_C,start ... i_C,end)] ← (Ψ†)[rows ∈ [i_C,start ... i_C,end)] · Ψ
    else
        ρ_L∘ ← ΨΨ†
        ρ_∘R ← Ψ†Ψ
    end if
end procedure
```

3.1.5 Construction of the Transformation Matrices

Having constructed the reduced density matrices, we proceed with obtaining their spectral decompositions which we will use to transform the spin and block operators for the next iteration. This requires the full spectral decomposition of relatively small dense matrices. Since the reduced density matrices are Hermitian and positive semi-definite, this is also equivalent to either an eigenvalue decomposition or singular value decomposition which are already implemented in SLEPc as an interface to dense LAPACK routines as well as its own iterative sparse solvers. The eigenpairs should then be sorted in descending order of the eigenvalue, either internally in the SLEPc routine or after the eigenpairs have been solved. This straightforward procedure is illustrated in Algorithm 6 and it is performed on the density matrices of both the left and right blocks.

Algorithm 6 Construction of the rotation matrices

Input:
Reduced density matrix ρ of size $dD \times dD$
m number of states to keep
Output: Rotation matrix $\mathbf{U}$, truncation error ε_{tr}
procedure GETROTATIONMATRICES
$\{(w_i, v_i)\}_{i=1}^{dD} \leftarrow$ eigenvalues and eigenvectors of ρ
Sort $\{(w_i, v_i)\}_{i=1}^{D}$ in descending order of w
$\varepsilon_{\text{tr}} \leftarrow 1 - \sum_{i=1}^{m} w_i$
$\mathbf{U} \leftarrow [v_1, v_2, \ldots, v_m]$
end procedure

3.1.6 Basis Truncation

The last step in the iteration is the rotation of enlarged block and site operators to the new basis. This is implemented in `TruncateOperators` named as such since the new basis is either of the same size or smaller than the original basis. The new operators are using $\tilde{\mathbf{O}} = \mathbf{U}^{\dagger}\mathbf{O}\mathbf{U}$ for all operators $\mathbf{O}$ in both the left and right blocks. This is implemented using the `MatMatMatMult` function in PETSc which does the triple matrix multiplication in parallel. In addition, there is also a function called `MatPTAP` which avoids building the explicit conjugate transpose but which is only applicable to the case of real scalars.

3.2 Quantum Number Conservation

Many of the computationally intensive operations in DMRG can be optimized by taking conserved quantum numbers into account. However, the basic implementation that we have described in the previous section does not require the explicit tracking of the basis that constitutes the blocks' Hilbert spaces. This means that taking conserved quantum numbers into account requires modifying the procedures discussed in our basic implementation to explicitly track the basis states and divide the full Hilbert space into sectors of similar quantum numbers. In this section, we detail how we implement $U(1)$ symmetry as discussed in Section 2.2.2, which in the context of the Heisenberg model refers to the conservation of total magnetization S^z_{tot}. The general structure of this implementation is adapted from the Simple-DMRG program written in Python [34].

3.2.1 Tracking of Magnetization Sectors

One of the first changes that introduced to our previous implementation is the tracking of the magnetization sectors in the blocks' Hilbert spaces. The magnetization values, which are eigenvalues of the $\mathbf{S}^z$ operator, are represented as an ordered tuple with entries that correspond to each state in our chosen matrix representation. For a block of one site containing one spin-$1/2$, the magnetization quantum numbers corresponding to the matrix representations in (2.13) form the ordered list

$$S^z_1 = \left[+\frac{1}{2}, -\frac{1}{2}\right] \tag{3.9}$$

This means that the Hilbert space contains two magnetization sectors containing one state each.

These conserved quantum numbers are also additive among the constituent sites of a block. Each time an enlarged block is produced, this list is expanded following an outer sum structure corresponding to the Kronecker product as shown in Algorithm 7. For example, when another site is added to (3.9), the enlarged block's magnetization list becomes

$$S^z_2 = [+1, 0, 0, -1] \tag{3.10}$$

which now contains three magnetization sectors, with $S^z = \pm 1$ containing one state

each, and $S^z = 0$ containing two states. This additional step is performed during the `BuildBlockLeft` and `BuildBlockRight` procedures. The magnetization sectors list is implemented in our code as a C++ standard vector for easy resizability.

On the other hand, we also need to track the magnetization sectors themselves by creating a map from a value of the magnetization to the corresponding set of indices of basis states that belong to that sector. We implement this as a C++ standard map that takes in a scalar value for the magnetization and a standard vector containing the indices of that sector. The subroutine, illustrated in Algorithm 8, will be useful later on when determining which sectors to pair up in constructing the superblock Hamiltonian with a target magnetization.

Algorithm 7 Performs the outer sum of quantum numbers between two blocks

Input: Quantum number lists S^z_A (size N_A) and S^z_B (size N_B)
Output: Quantum number list S^z_{AB} of the combined block
  procedure OUTERSUMFLATTEN
    for $i \in [0, N_A)$ **do**
      for $j \in [0, N_B)$ **do**
        $S^z_{AB}[i \cdot N_B + j] \leftarrow S^z_A[i] \cdot S^z_B[j]$
      end for
    end for
  end procedure

Algorithm 8 Creating the map from quantum number to sector indices

Input: Magnetization list S^z with N entries
Output: Mapping object SECTORINDICES
  procedure INDEXMAP
    Prepare SECTORINDICES: float $\rightarrow$ vector of integers
    for $i \in [0, N)$ **do**
      Append i to SECTORINDICES$[S^z[i]]$
    end for
  end procedure

3.2.2 Solving for the Ground State in a Target Sector

When solving for a specific state, one can often find it in a specific quantum number sector. For example, the ground state in a spin-$1/2$ Heisenberg chain is the ground state in the magnetization sector $S^z_{\text{tot}} = 0$, while the first excited state is the ground state in $S^z_{\text{tot}} = +1$. Also, the most time-consuming and memory intensive steps in the DMRG algorithm are the construction and diagonalization of the superblock Hamiltonian due to the large size of the matrix which has a lot of basis states. Thus, tracking the sectors of each state allows us to use only those states that belong to our target sector when constructing the Hamiltonian, thereby reducing the computational cost involved.

We can use the INDEXMAP procedure in Algorithm 8 to group together the states that belong to a specific sector. Then, we can select which sectors from the left and right blocks are to be paired together and determine the Kronecker product indices which belong to the target sector. This additional step as shown in Algorithm 9 is performed at the beginning of the `BuildSuperBlock` procedure. The index list V will be used to select which states belong to the target sector in the superblock Hamiltonian. A mapping M'_L is also created to track the sectors of the resulting ground state vector.

Algorithm 9 Selection of states in the target sector

Input: Magnetization lists S^z_L and S^z_R,
Target magnetization sector S^z_{tot}
Output: List of indices V of states belonging to the superblock's target sector,
Map M'_L of left block's sector indices in the superblock
procedure SELECTSECTORSTATES
$M_L \leftarrow$ INDEXMAP(S^z_L)
$M_R \leftarrow$ INDEXMAP(S^z_R)
for $(s_L, V_L) \in M_L$ **do** ▷ s_L is an S^z value and V_L are sector indices
$s_R \leftarrow S^z_{\text{tot}} - s_L$
$V_R \leftarrow M_R[s_R]$
for $i \in V_L$ **do**
for $j \in V_R$ **do**
Append $(i \cdot N_R + j)$ to V
Append length(V) to $M'_L[s_L]$
end for
end for
end for
end procedure

3.2.3 Kronecker Product with Index Slicing

In constructing the superblock Hamiltonian with a restricted basis, we have to modify the Kronecker product routine itself. First of all, a new input list V is introduced containing the basis indices produced in Algorithm 9. We use these indices to perform array slicing on both the rows and columns of the full Kronecker product matrix. However, to reduce memory and communication, we avoid building the full matrix itself. Taking the CSR storage format of our matrices into account, we perform the array slicing on the rows first since this can be done much more easily. This procedure is demonstrated in Algorithm 10 as a modification to Algorithm 2 by looping through the indices in V when performing the submatrix collection.

Having the needed submatrices on each MPI process, we build an intermediate submatrix which reflects the row-sliced superblock Hamiltonian. We preallocate the memory needed for the intermediate sequential submatrix as shown in Algorithm 11 which is much simpler compared to the preallocation for the global matrix in Algorithm 3 since local submatrices are not divided into diagonal and off-diagonal blocks.

The entries for each row are then explicitly constructed into an intermediate sequential matrix C' in the same manner as in Algorithm 4. We can now perform the column slicing on each process' sequential matrix C' using PETSc's `MatGetSubmatrices` function. These matrices are then stitched together into a single global matrix using `MatCreateMPIMatConcatenateSeqMat` which handles everything including preallocation. We show the details of these steps in Algorithm 12. [23, 29]

Algorithm 10 Indexed Kronecker Product: Submatrix Collection

Input:
MPI matrices $\{\mathbf{A}_n\}_{n=1}^{M}$ of size (M_A,N_A) and $\{\mathbf{B}_n\}_{n=1}^{M}$ of size (M_B,N_B)
Indices V of the restricted basis
MPI communicator rank n_{rank} and size n_{procs}
Output: Sequential matrices $\{\mathbf{A}_{n,\text{sub}}\}_{n=1}^{M}$ and $\{\mathbf{B}_{n,\text{sub}}\}_{n=1}^{M}$ containing non-local rows
procedure KRONIDXGETSUBMATRICES
$M_C \leftarrow \text{length}(V)$ ▷ Global number of rows of restricted **C**
$N_C \leftarrow N_A \cdot N_B$ ▷ Global number of columns of full **C**
$i_{C,\text{start}}, i_{C,\text{end}} \leftarrow$ SPLITOWNERSHIPRANGE(M_C, n_{rank}, n_{procs})
for $i_C \in \left[i_{C,\text{start}}, i_{C,\text{end}}\right)$ **do**
$\text{IS}_A \leftarrow \text{insert}(V[i_C]/M_B)$ ▷ Index set of required rows of **A**
$\text{IS}_B \leftarrow \text{insert}(V[i_C] \bmod M_B)$ ▷ Index set of required rows of **B**
end for
for $n \in \{1 \ldots M\}$ **do**
$\mathbf{A}_{n,\text{sub}} \leftarrow$ `MatGetSubmatrices` of all rows of $\mathbf{A}_n$ listed in IS_A
$\mathbf{B}_{n,\text{sub}} \leftarrow$ `MatGetSubmatrices` of all rows of $\mathbf{B}_n$ listed in IS_B
end for
end procedure

Algorithm 11 Indexed Kronecker Product: Preallocation of Intermediate Matrix

Input: Sets of submatrices $\{\mathbf{A}_{n,\text{sub}}\}_{n=1}^{M}$ and $\{\mathbf{B}_{n,\text{sub}}\}_{n=1}^{M}$
Output: Array NNZ[] containing the number of non-zeros in the sequential matrix
procedure KRONIDXPREALLOCATION
for $i_C \in \left[i_{C,\text{start}}, i_{C,\text{end}}\right)$ **do**
$\text{NNZ}[i_C] \leftarrow 0$
for $n \in \{1 \ldots M\}$ **do**
$\text{NNZ}_A \leftarrow \text{length}(\mathbf{A}_{n,\text{sub}}[i_C/M_B])$
$\text{NNZ}_B \leftarrow \text{length}(\mathbf{B}_{n,\text{sub}}[i_C \bmod M_B])$
$\text{NNZ}[i_C] \leftarrow \text{NNZ}[i_C] + \text{NNZ}_A \cdot \text{NNZ}_B$
end for
end for
end procedure

Algorithm 12 Indexed Kronecker Product: Matrix Construction

Input: Coefficients $\{a_n\}$ of the linear combination of $\otimes$
Sets of local submatrices $\{\mathbf{A}_{n,\text{sub}}\}_{n=1}^{M}$ and $\{\mathbf{B}_{n,\text{sub}}\}_{n=1}^{M}$ containing non-local rows
Output: Resultant matrix $\mathbf{C}$

procedure KRONIDXMATRIXCONSTRUCTION
  for $\text{row}_C \in \left[i_{C,\text{start}}, i_{C,\text{end}}\right)$ **do**
    $\text{row}_A \leftarrow V[i_C]/M_B$
    $\text{row}_B \leftarrow V[i_C] \bmod M_B$
    for $n \in \{1 \dots M\}$ **do**
      for $(\text{col}_A, \text{val}_A) \in \mathbf{A}_{n,\text{sub}}[\text{row}_A]$ **do**
        for $(\text{col}_B, \text{val}_B) \in \mathbf{B}_{n,\text{sub}}[\text{row}_B]$ **do**
          $\text{col}_C \leftarrow \text{col}_A \cdot N_B + \text{col}_B$
          $\text{val}_C \leftarrow a_n \cdot \text{val}_A \cdot \text{val}_B$
          $\mathbf{C}'[\text{row}_C] \leftarrow \text{insert}\left((\text{col}_C, \text{val}_C)\right)$ ▷ Row-sliced seq. matrix
        end for
      end for
    end for
  end for
  for $j_C \in V$ **do**
    $\text{IS}_C \leftarrow \text{insert}(j_C)$ ▷ Index set for columns
  end for
  $\mathbf{C}'' \leftarrow$ `MatGetSubmatrices` of all columns of $\mathbf{C}'$ listed in IS_C
  $\mathbf{C} \leftarrow$ `MatCreateMPIMatConcatenateSeqMat` of $\mathbf{C}''$ in all processes
end procedure

3.2.4 Block Diagonal Reduced Density Matrices

The iterative diagonalization of the superblock Hamiltonian using SLEPc's EPS class proceeds in the same manner as in Listing 3.1. Additionally, we now have explicit information on the quantum number of each of the ground state vector's indices provided by M'_L in Algorithm 9. By fixing the total magnetization, the corresponding reduced density matrices acquire a block-diagonal structure [15]. Thus, instead of diagonalizing a reduced density matrix of size $dD \times dD$, we can instead work on its smaller diagonal blocks. The construction of the block diagonal reduced density matrices are shown in Algorithm 13. It uses the same intermediate matrix Ψ as in Algorithm 5 to build each block diagonal but with a smaller dimension of $N_L \times N_R$

The spectral decomposition can also be implemented on each diagonal block of the reduced density matrices as shown in Algorithm 14 which involves significantly more steps than Algorithm 6 but is more computationally efficient since it involves the eigendecomposition of smaller matrices. The final truncation step of all operators proceeds in the same manner as the basic implementation using $\tilde{\mathbf{O}} = \mathbf{U}^\dagger \mathbf{O} \mathbf{U}$ for all operators $\mathbf{O}$ in both the left and right blocks.

Algorithm 13 Construction of the block-diagonal reduced density matrices

Input:
Global ground state vector ψ_{GS},
Sector indices map of the superblock Hamiltonian M'_L,
Sector indices maps of the left and right blocks M_L and M_R,
Target magnetization S^z_{tot}

Output:
S^z sectors and reduced density matrices $\{(s^q_L, \hat{\rho}^q_{L\circ})\}_{q=1}^{Q_L}$ and $\{(s^q_R, \hat{\rho}^q_{\circ R})\}_{q=1}^{Q_R}$

procedure BUILDBLOCKDIAGONALREDUCEDDENSITYMATRICES
  $\psi \leftarrow$ full local copy of ψ_{GS} using `VecScatter`
  $q \leftarrow 0$ ▷ Quantum number counter
  for $(s_L, V_L) \in V'_L$ **do**
    $s_R \leftarrow S^z_{\text{tot}} - s_L$
    $N_L \leftarrow \text{length}(M_L[s_L])$ ▷ No. of states in the sector on the left block
    $N_R \leftarrow \text{length}(M_R[s_R])$ ▷ No. of states in the sector on the right block
    if $N_L \cdot N_R = 0$ **then** Continue
    end if
    Prepare $N_L \times N_R$ local matrix Ψ
    for $i \in [0, N_L)$ **do**
      for $j \in [0, N_R)$ **do**
        $\Psi[i, j] \leftarrow \psi[i \cdot N_R + j]$ ▷ Copy values in row-major order
      end for
    end for
    $(s^q_L, \hat{\rho}^q_{L\circ}) \leftarrow (s_L, \hat{\Psi}\hat{\Psi}^\dagger)$
    $(s^q_R, \hat{\rho}^q_{\circ R}) \leftarrow (s_R, \hat{\Psi}^\dagger\hat{\Psi})$
    $q \leftarrow q + 1$
  end for
end procedure

Algorithm 14 Construction of the rotation matrices

Input:
$\{(s^q, \hat{\rho}^q)\}_{q=1}^{Q}$ sectors and diagonal blocks of a reduced density matrix ρ,
M sector indices map of the corresponding block,
m number of states to keep
Output:
Rotation matrix $\mathbf{U}$,
truncation error ε_{tr}
procedure GETBLOCKDIAGONALROTATIONMATRICES
Let $\{D_q\}_{q=1}^{Q}$ number of states in each diagonal block
$D \leftarrow \sum_{q=1}^{Q} D_q$ total number of states of ρ
Prepare $\Lambda \equiv \{(w_i^q, v_i^q)\}_{i=1}^{D}$ spectral decomposition of ρ
for $q \in \{1, 2, \ldots, Q\}$ **do**
$\Lambda_q \leftarrow \{(w_i^q, v_i^q)\}_{i=1}^{D_q}$ spectral decomposition of sub-block ρ^q
Append Λ_q to Λ
end for
Sort Λ in descending order of w and keep only the first m entries
Truncation error: $\varepsilon_{\text{tr}} \leftarrow 1 - \sum_{i=1}^{m} w_i$
Prepare $\mathbf{U}$ of dimension $D \times m$
for $(w_i^q, v_i^q) \in \Lambda$ **do**
for $j \in M[s^q]$ **do**
$\mathbf{U}_{j,i} \leftarrow (v_i^q)_j$
end for
end for
end procedure

CHAPTER 4

Results and Performance

In this chapter we evaluate the performance and scalability of our DMRG application. In Section 4.1, we describe the target hardware architecture and the compilation of software dependencies on which we run our numerical calculations. In Section 4.2 we briefly look at how we evaluate the accuracy of our numerical results. Then, we present the results of our performance measurements in Section 4.3, followed by the discussion of these results in Section 4.4.

4.1 Computational Tools

4.1.1 System Architecture

The systems we targeted with our DMRG implementation are two partitions of CINECA Marconi which is a class Tier-0 supercomputer that runs on the Intel Xeon family of processors. Specifically, we perform our numerical experiments on the Marconi A1 partition which runs on Broadwell (BDW) processors, and on the Marconi A2 partition which runs on Knights Landing (KNL) or Xeon Phi processors. Details on the relevant specifications of each partition are provided in Table 4.1.

From the table, several key differences between BDW and KNL are evident. KNL nodes have a higher core count of slower processors, but with a higher peak performance compared to BDW. The BDW nodes have more total memory than KNL, but the KNL processors have an on-chip high-bandwidth memory of 16 GB offered by MCDRAM on top of the regular DDR4. The production nodes on KNL use MCDRAM in cache mode, in which they function as L3 cache that is invisible to the user, instead of acting

Table 4.1: Details of the CINECA Marconi BDW and KNL partitions [35]

	Marconi A1 (BDW)	**Marconi A2 (KNL)**
Processors	2 × 18-core Intel Xeon E5-2697 (Broadwell) at 2.3 GHz	68-core Intel Xeon Phi 7250 CPU (Knights Landing) at 1.40 GHz
Cores	36 cores/node; 54,432 cores total	68 cores/node; 244,800 cores total
Instruction Set Extensions	AVX 2.0	AVX-512
RAM	128 GB/node	16 GB/node of MCDRAM and 96 GB/node of DDR4
Max Memory Bandwidth	76.8 GB/s	115.2 GB/s
Network	Intel Omnipath, 100 Gb/s	

as a separate physical address space. [35]

BDW processors work on the standard 64-bit instruction set with Advanced Vector Extensions 2.0 (AVX2) extensions which enable 256-bit wide vectorizations. On the other hand, KNL has added support for AVX-512, which expands the width of AVX2 vectorization to 512 bits. This means that, in general, binaries compiled for the BDW processors are also compatible for KNL, but not vice versa.

4.1.2 Software Dependencies and Configuration

Our DMRG application was compiled with several software libraries tailored for Intel processors to ensure maximum possible performance on our target computing systems. For performance tests on BDW and KNL, we used PETSc version 3.7.2 and SLEPc version 3.7.3, which are provided as modules on Marconi, and which were built using Intel MPI compiler and Intel Math Kernel Library (MKL) 2017 for linear algebra routines. It was configured with the default 64-bit real scalar values and 32-bit integers, and with Fortran kernels enabled for array operations.

The DMRG application that we have written was compiled using Makefiles with directives to include configuration files from the targeted build of PETSc and SLEPc. The C++ files were compiled with the `-std=c++11` flag indicating the standard, and the `-O3` optimization flag.

4.2 Convergence of Ground State Energy Per Site

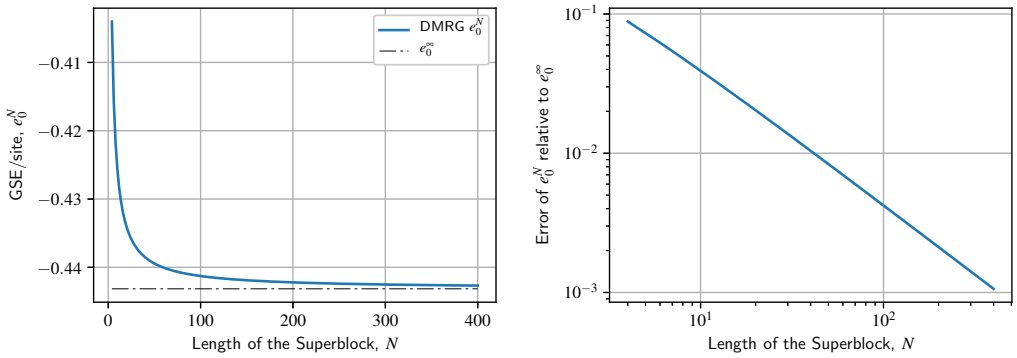

Figure 4.1: *Left*: Ground state energy per site $e_0^N(m)$ for different number of sites N of the superblock obtained during a single infinite-size DMRG run. *Right:* Corresponding error of $e_0^N(m)$ relative to the known value at the thermodynamic limit e_0^∞. Values were obtained with $m = 512$ kept states.

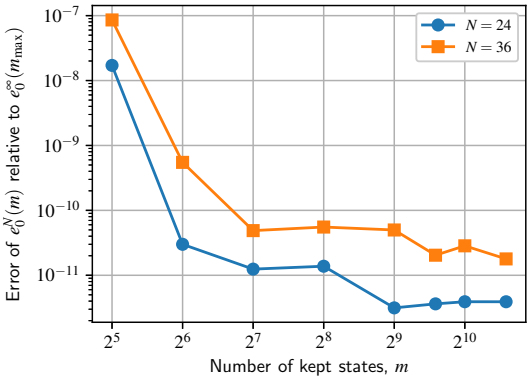

Figure 4.2: Convergence of $e_0^N(m)$ for increasing number of kept states. The error was calculated with respect to $e_0^N(m_{\max} = 2048)$.

We test the accuracy of our DMRG implementation by tracking the ground state energy per site of the spin-$1/2$ Heisenberg chain obtained from the diagonalization of the superblock Hamiltonian at each iteration. There are two ways to evaluate accuracy and convergence. First, the ground state energy per site $e_0^N(m)$ for N sites and m kept states should converge towards the known result at the thermodynamic limit $e_0^\infty = 0.4431...$ although the calculated energy will always be a bit larger due to finite-size corrections [36]. The convergence of the energy is shown in Figure 4.1. Second, the values of $e_0^N(m)$ should also converge as the number of kept states increases. This is shown in Figure 4.2 in which $e_0^N(m)$ converges towards the value obtained at the largest number of kept states $e_0^N(m_{\max})$.

4.3 Performance Analysis

In this section, we present results of the performance measurements of our DMRG implementation. The numerical calculations were done on the Marconi A1 (BDW) and A2 (KNL) clusters. Elapsed time and memory consumption were measured using PETSc profiling tools from the root MPI process (rank 0) and scaled accordingly. All DMRG simulations presented here involve growing a superblock chain of $N = 36$ sites with various number of kept states after truncation. At each iteration, we calculate the ground state which resides in the magnetization sector $S^z_{\text{tot}} = 0$. Elapsed times are shown color-coded for each component step of our workflow (Section 3.1.1), while memory consumption is shown for the objects with the most memory usage.

4.3.1 Comparison Between Implementations

We first compare the two DMRG implementations that we have detailed in the previous chapter: the basic implementation without symmetries, and the implementation with $U(1)$ symmetry or conservation of total magnetization S^z_{tot}. We look at the resulting improvement from the basic implementation which solves for the ground state from among all magnetization sectors, to the second implementation which targets the ground state in the $S^z_{\text{tot}} = 0$ sector.

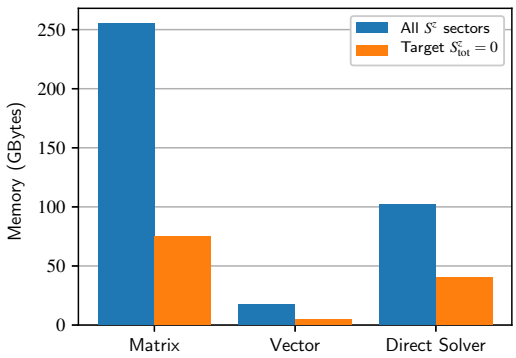

Figure 4.3: Comparison of memory usage among different PETSc objects between the basic implementation without symmetries targeting all S^z_{tot} sectors and the implementation which targets the $S^z_{\text{tot}} = 0$ sector.

As shown in Figure 4.3, memory consumption is significantly reduced in the second implementation. This is due to the fact that the $S^z_{\text{tot}} = 0$ sector is a smaller subset of all possible states of the superblock. This results in a smaller Hamiltonian matrix which in turn produces smaller vectors, and which is diagonalized with less memory usage.

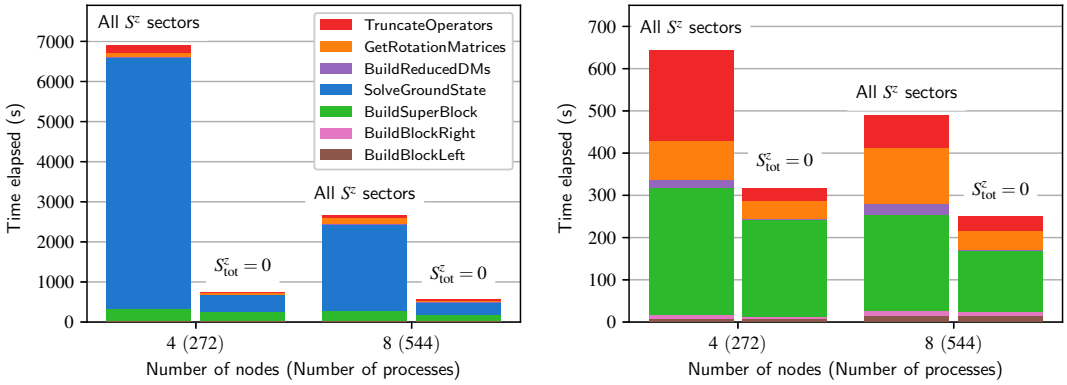

Figure 4.4: Comparison of elapsed time of simulations with $m = 1024$ kept states between the basic implementation without symmetries targeting all S^z_{tot} sectors and the implementation which targets the $S^z_{\text{tot}} = 0$ sector. *Left*: All steps in the algorithm; *Right:* The same measurements showing all steps except SolveGroundState.

From the plots in Figure 4.4, we can see that in both implementations the most time-consuming steps are the construction of the large superblock Hamiltonian (BuildSuperBlock), its iterative diagonalization (SolveGroundState), the eigendecomposition of the reduced density matrices (GetRotationMatrices) and the rotation of the operators to the truncated basis (TruncateOperators). In the implementation with a target magnetization, we can immediately see an order of magnitude improvement in the time to solution. The largest improvement comes from SolveGroundState as it also benefits from the smaller size of the Hamiltonian matrix.

Also, there is significant improvement in the elapsed time for the last three steps which involve operations on smaller matrices. The BuildReducedDMs and GetRotationMatrices steps are faster because of the block diagonal structure of the reduced density matrices $\rho_{L\circ(\circ R)}$ since tracking magnetization sectors allows us to work only on several smaller diagonal blocks, instead of a full matrix. The TruncateOperators speeds up the most since the transformation matrices $\mathbf{U}_{L(R)}$ which were constructed dense in the basic implementation become sparse and block diagonal in the second.

During the BuildSuperBlock step, in which the superblock Hamiltonian is constructed, array slicing for rows and columns corresponding to the target sector states are done as separate steps to reduce memory consumption. This means that within this step intermediate matrices have to be created and destroyed, and values have to be accessed and copied between these matrices. Thus, although overall matrix memory consumption is significantly reduced, in terms of elapsed time this step only slightly improves in the second implementation.

Since the the implementation with $U(1)$ symmetries gives a very good advantage in terms of solution time and memory consumption, it will be used in all numerical simulations presented in the following sections.

4.3.2 Strong Scalability

In this section, we evaluate the strong scalability of our DMRG code; that is, we quantify the improvement in the time to solution for a fixed problem size as the number of computational resources such as processors and nodes are increased. We can measure strong scalability using *speedup* and *parallel efficiency*. The speedup s_p is defined as the ratio

$$s_p = \frac{t_1}{t_p} \tag{4.1}$$

where t_p is the elapsed time for p processing elements (nodes or cores), and t_1 is the the best time with 1 processing element. Ideal speedup of $s_p = p$ occurs when additional resources produce a directly proportional improvement in performance. In some instances t_1 is intractable to acquire directly, for example when the problem size doesn't fit on the memory of one node; instead, we use $t_1 = qt_q$ for some baseline number of processing elements q. Parallel efficiency, on the other hand, is the ratio between the obtained speedup and the ideal speedup, and it is given by

$$e_p = \frac{s_p}{p} = \frac{t_1}{p\, t_p} \tag{4.2}$$

with an ideal value of $e_p = 1$. [37]

We first look at the case of a few processors and problem sizes with $m = 512,\ 768$ kept states whose memory requirements can fit on one node. Figure 4.5 shows the elapsed time for different values of m and for the two sections of Marconi. We can immediately see that runs performed on BDW are generally faster than those on KNL when comparing among the same number of processes.

From these run times, we obtain the speedup and efficiency as shown in Figure 4.6. Although the BDW cores are faster, results from KNL show better scalability. If we also look at the parallel efficiency of the most time-consuming steps of the algorithm, namely BuildSuperBlock and SolveGroundState, both algorithms show similar scaling behavior since they are both memory-bound operations. Thus, the most probable reason

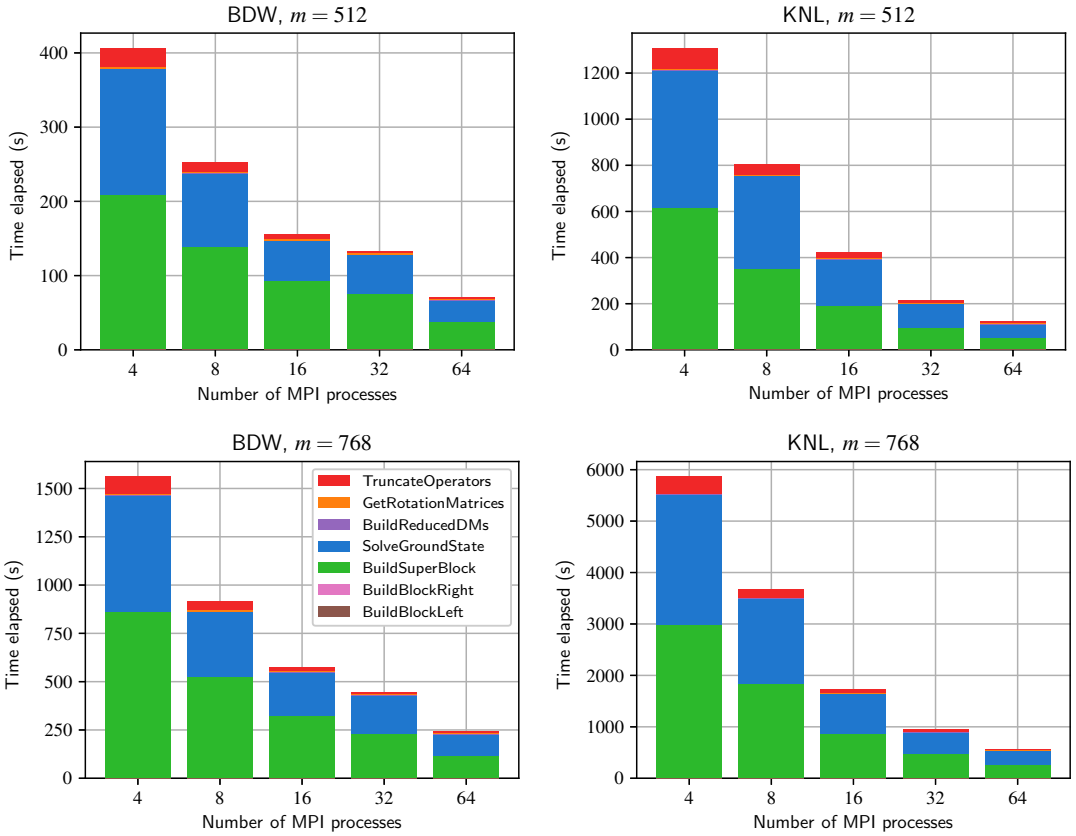

Figure 4.5: Elapsed time for DMRG runs with $m = 512$ (top) and $m = 768$ (bottom) kept states showing strong scaling behavior of the program on BDW (left) and KNL (right).

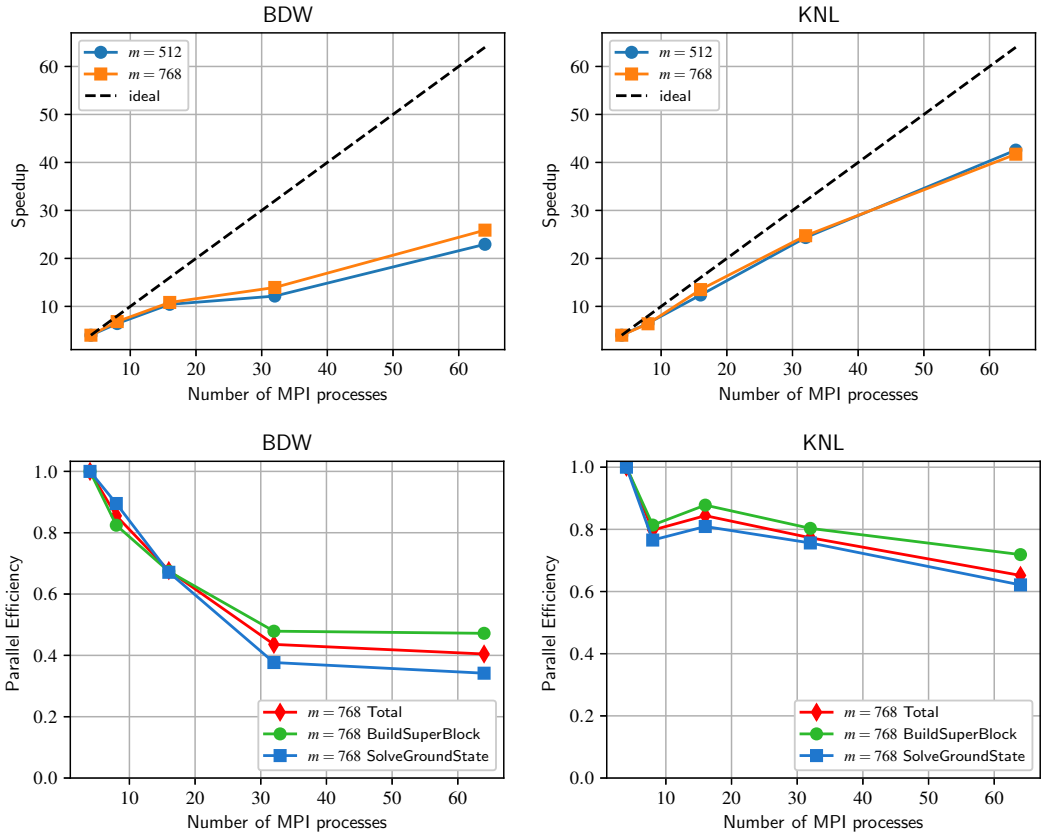

Figure 4.6: Strong scalability for DMRG runs on up to two nodes of BDW (left) and a single node of KNL (right). *Top:* Speedup for two different values of the number of kept states m. *Bottom:* Parallel efficiency for $m = 768$, also showing the most time consuming steps: the construction of the superblock Hamiltonian (BuildSuperBlock) and its iterative diagonalization (SolveGroundState).

for better scalability on KNL processors than on BDW is that KNL nodes have higher memory bandwidth which are able to handle more processors that are simultaneously performing memory-bound operations. This is true for most sparse matrix operations whose performance is almost always determined by memory bandwidth instead of the speed of CPU.*

Next, we explore cases where the the number of kept states m results in memory requirements that exceed a single node. In these cases, we consider the node as a single processing element with several processors per node (ppn). Also, we compare two cases for the number of processors used on each node:

- *full-node case* when we run the application using all available ppn; and
- *half-node case* when using only half the available ppn.

This comparison is motivated by two reasons. First, each processor in the half-node case gets twice the memory bandwidth that they get in the full-node case. Since the parallel efficiency of our application is highly limited by memory bandwidth, we expect that reducing the number of processors per node should help prevent memory bandwidth saturation and improve overall performance. Second, PETSc developers recommend keeping a high number of unknowns per process so that computation time outweighs the latency in communication.

Figure 4.7 shows the case of $m = 1024$ kept states starting at 2 nodes. We see a similar proportion of the time elapsed for each section as in the previous cases with smaller m. We also observe here that for the same number of nodes, simulations on BDW are faster even though it has less processors than KNL. By using only half the maximum ppn, the elapsed times may be higher for smaller number of nodes, but there is a pay off at 8 and 16 nodes especially for KNL; in these cases, using less ppn is faster than using all available processors.

The steps that act on the large Hamiltonian matrix, namely BuildSuperBlock and SolveGroundState show good scaling behavior. However, the steps that act on smaller matrices, namely BuildReducedDMs, GetRotationMatrices and TruncateOperators, do not significantly scale at all since the processes performing the tasks are already too many for the size of these matrices. The size discrepancy among our matrices results from the fact that the superblock Hamiltonian matrix scales as m^2 while the size of the reduced density matrices and the transformation matrices scale only linearly with m.

Figure 4.8 shows the corresponding parallel efficiencies. In all cases, the BuildSu-

*See PETSC FAQ entry on Why do I get little speedup...? [29]

perBlock step shows good scaling in performance since communication is required only at the beginning during submatrix collection and at the end of the step during the final matrix assembly. However, the improvement in the performance of the SolveGround-State step diminishes much earlier at 8 nodes; this is because increasing the number of nodes increases the latency in communication of vector elements required during the many sparse matrix-vector multiplications in the iterative diagonalization routines. This is reflected by the similar scaling pattern between the full-node case in BDW and the half-node case of KNL which have almost the same total number of processes per node. In the half-node case, we observe improved parallel efficiency on both clusters compared to the full-node case due to better utilization of the memory bandwidth and increase in computation time versus communication. This is especially true for BDW since it has a lesser core count per node.

We go higher and consider the simulations for $m = 1536$ kept states which manage to fit starting at 4 nodes of BDW and 6 nodes of KNL. From the elapsed times and parallel efficiencies shown in Figures 4.9 and 4.10, respectively, the same general observations regarding scaling can also be made as in $m = 1024$. The BuildSuperBlock step continues to scale well, as expected. However, the scaling problems in SolveGroundState are now magnified; for the full-node cases, its time elapsed on both BDW and KNL stop scaling after 12 nodes and worsens as more nodes are added. This is due to latency in communication during the matrix-vector products which can be solved by going into the half-node case. In the half-node case of BDW, however, we see scaling only up to 16 nodes and it worsens from there. On the other hand, we do not see this problem on KNL which even shows a "super-scaling"effect in some points which can be attributed to KNL's higher memory bandwidth per processor and cache effects from MCDRAM.

We also consider $m = 2048$ kept states whose memory requirements fit on 12 nodes of BDW and KNL. We now consider only the half-node case since it has shown superior performance and scalability for larger number of nodes. Figure 4.11 shows that BDW scales poorly and saturates after 24 nodes, while KNL continues to scale up to 32 nodes and levels off at 40 nodes. In Figure 4.12 KNL even exhibits superscaling behavior which may also be attributed to cache effects from MCDRAM.

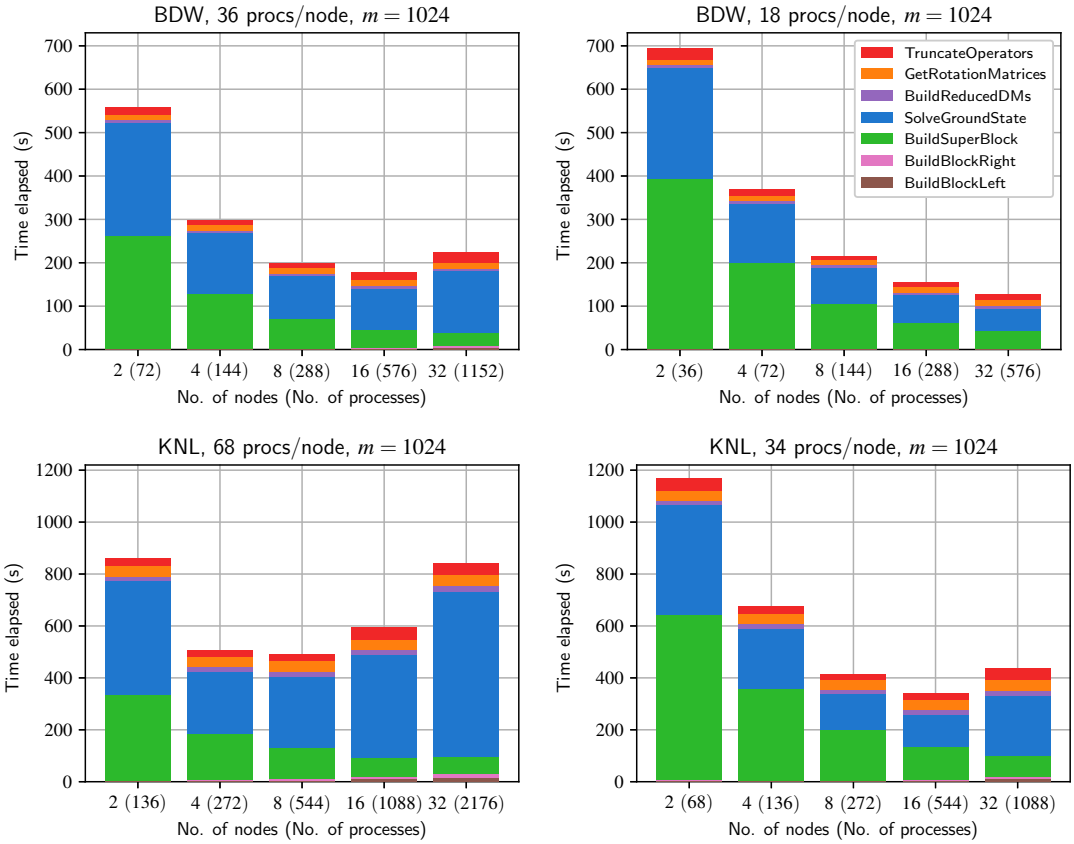

Figure 4.7: Elapsed time for DMRG runs with $m = 1024$ kept states on multiple nodes of Marconi BDW (top) and KNL (bottom) for the full-node case (left) and for the half-node case (right).

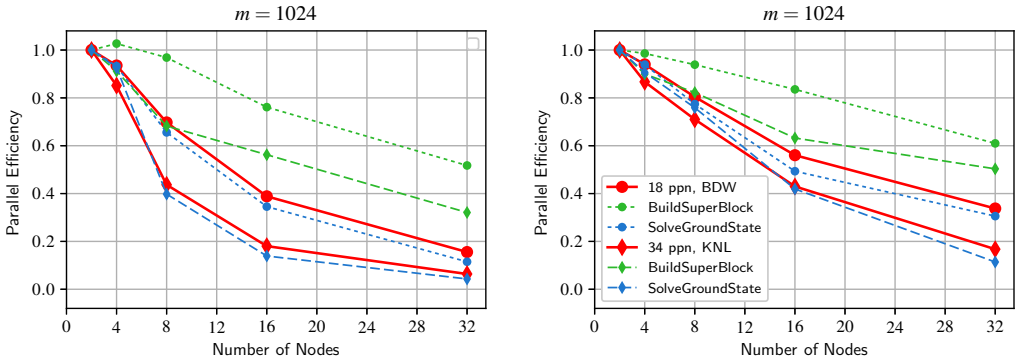

Figure 4.8: Parallel efficiency for DMRG runs with $m = 1024$ kept states for the full-node case (left) and for the half-node case (right).

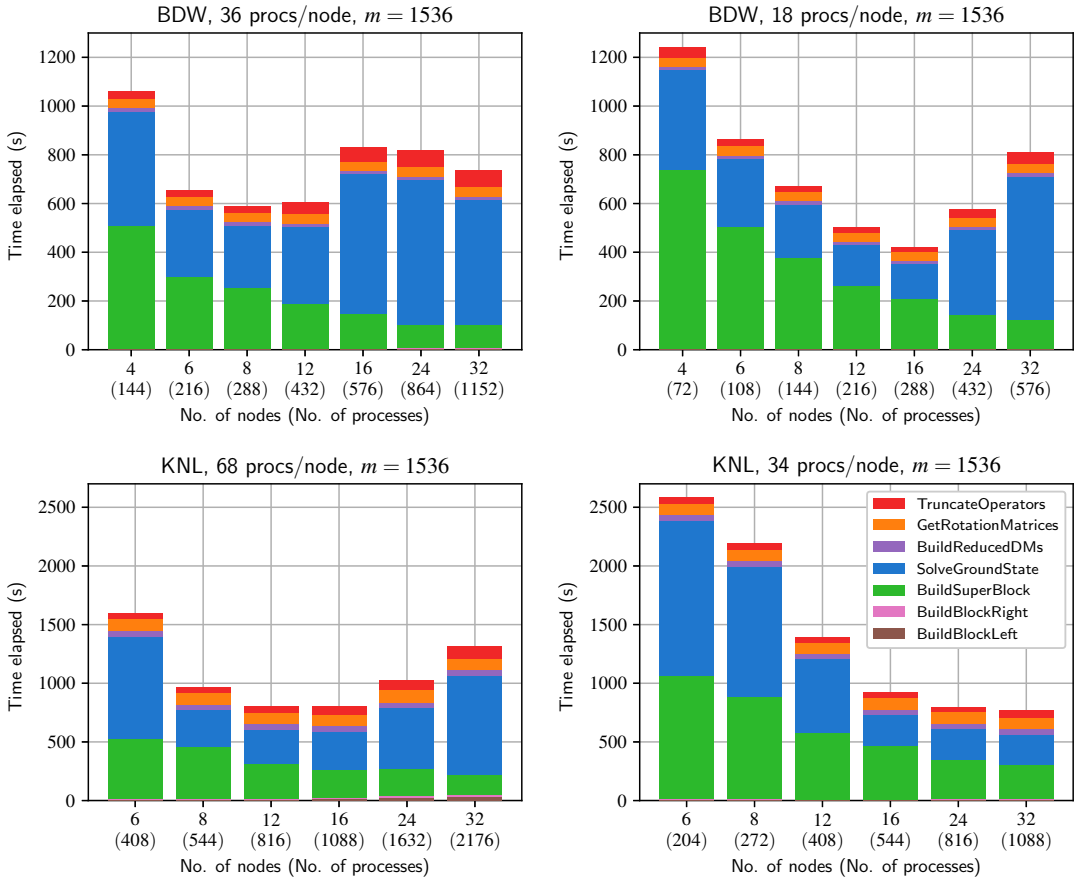

Figure 4.9: Elapsed time for DMRG runs with $m = 1536$ kept states on multiple nodes of Marconi BDW (top) and KNL (bottom) for the full-node case (left) and for the half-node case (right).

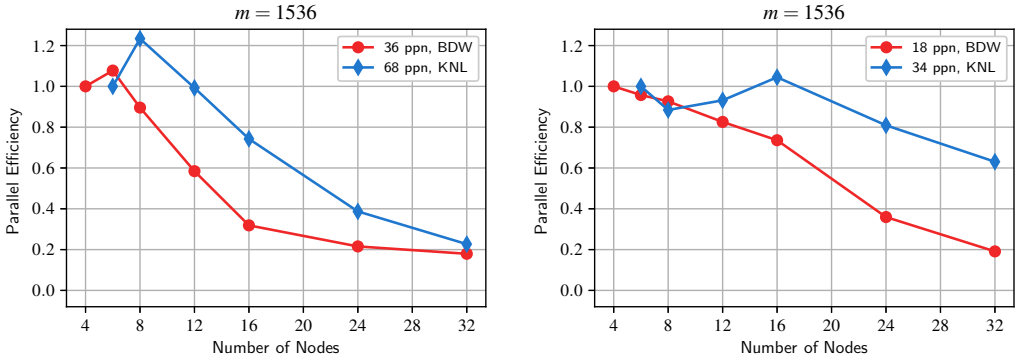

Figure 4.10: Parallel efficiency for DMRG runs with $m = 1536$ kept states for the full-node case (left) and for the half-node case (right).

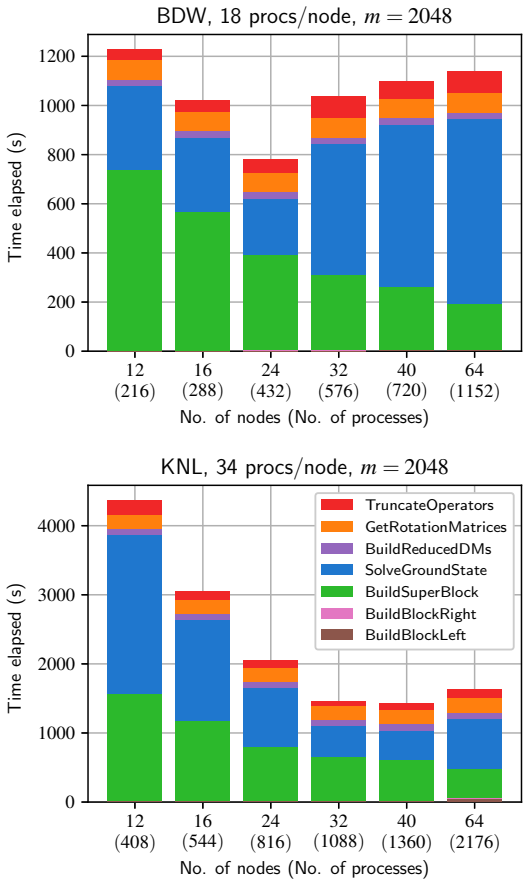

Figure 4.11: Elapsed time for DMRG runs with and $m = 2048$ kept states on multiple nodes of Marconi BDW (top) and KNL (bottom) running on half of the available processors on each node.

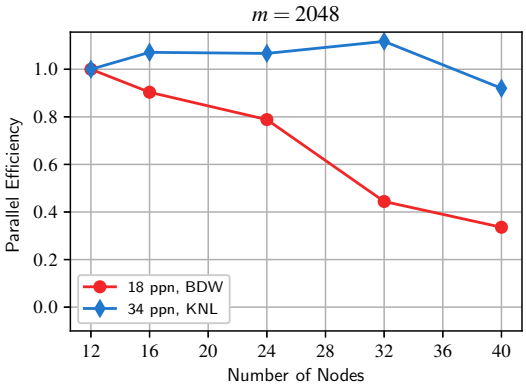

Figure 4.12: Parallel efficiency for DMRG runs with $m = 2048$ kept states for the half-node case.

4.3.3 Scaling of Computational Resources with Problem Size

From the previous section, we noted that for a given number of kept states m, there is a particular number of nodes that the problem size can fit in memory. Thus, if we want to target a higher value of m, we need to quantify the computational resources required for our DMRG simulation by extrapolating our available data. Figures 4.13-4.14 show the scaling for the elapsed time for each iteration and the memory consumption. Both variables exhibit regular monomial-like behavior which are quantified by the good linear fit on the log-transformed data. The trendline equations provide a way to predict how many nodes would be needed and how much time should be requested for different values of m especially when running the simulations through a job scheduler.

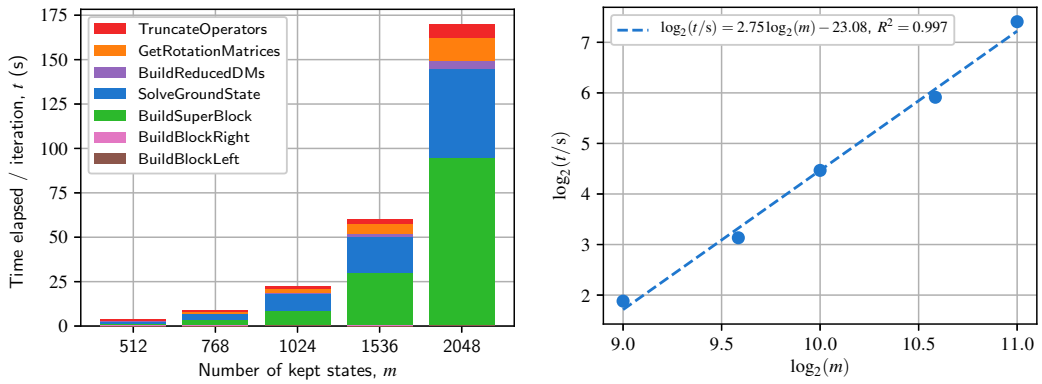

Figure 4.13: Scaling behavior of elapsed time per iteration, t, for varying number of kept states m on 16 nodes of Marconi BDW running at half node.

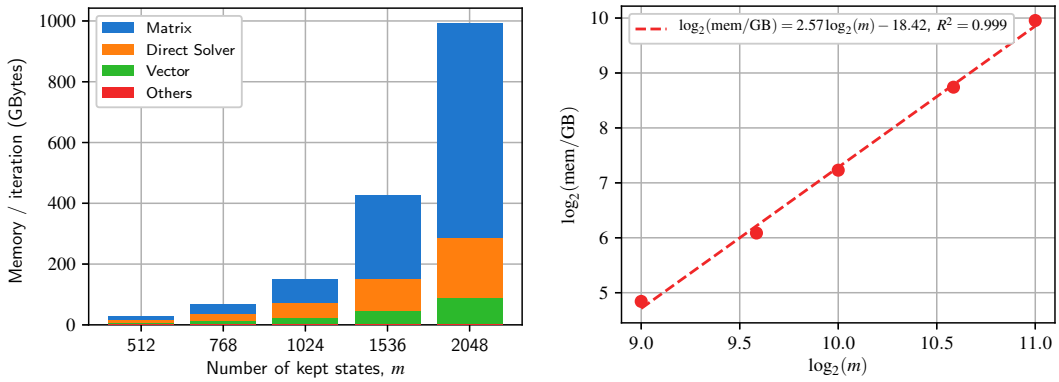

Figure 4.14: Scaling behavior of memory consumption per iteration, *mem*, for varying number of kept states m.

4.4 Discussion

We have shown from the performance results that our DMRG implementation generally exhibits good scaling behavior especially with the most time-consuming parts of the algorithm which involve the large superblock Hamiltonian. The construction of the Hamiltonian in the BuildSuperBlock step scales very well in all cases since communication among processes occur only at the beginning during submatrix collection and at the end during the final matrix assembly. However, overall scalability is limited by the SolveGroundState step which is the iterative diagonalization of the superblock Hamiltonian. We have identified three key factors that limit the scalability of SolveGroundState.

First, it is dependent on the performance of SLEPc's eigensolver which in turn is limited by the efficient parallel sparse matrix-vector multiplications (spMVM). This means we need to keep the number of unknowns on each process high enough such that the time for computation overcomes the time for the all-to-all communication of vector elements [31]. The developers of PETSc suggest that each process should hold a minimum of $10,000$ unknowns or local rows of the matrix, and recommend at least $20,000$ or more.[†] Taking the case of $m = 1536$ whose superblock matrix has an average of 4.4×10^6 global rows, this translates to a range of around 220 to at most 440 processors. For this reason, we see a deterioration in scalability at this point in Figure 4.9 which is more pronounced on BDW than on KNL.

Second, different problem sizes, parametrized by the number of kept states m, require different minimum numbers of nodes making it difficult to directly compare their scaling behaviors; however, we see a general trend that increasing the problem size benefits overall scalability on the same set of number of nodes.

Third, the performance of spMVM is also limited by the memory bandwidth per core. Thus we have seen that operating in the half-node case always improves overall scaling and even results in faster time-to-solution for higher number of nodes. This is clearly because in the half-node case, the memory bandwidth available to each process is double that in the full-node case.

Thus, to achieve the best performance for a desired m, one has to look for the configuration with (1) high enough total number of processes to push the limits of time for computation versus communication, (2) enough number of nodes to fit the problem in memory, and (3) a good minimum number of processes per node so that each process

[†]See PETSC FAQ entry on Why is my parallel solver slower than my sequential solver...? [29]

still gets high memory bandwidth.

Moreover, the steps involving the block operators, the reduced density matrix ρ and the transformation matrix $\mathbf{U}$ do not scale since the number of processes needed to efficiently perform the steps for the large superblock Hamiltonian is already too many for these small matrices. This may be addressed by implementing a different level of parallelism for the smaller matrices.

In the results that we have presented, we have also made comparisons between the BDW and KNL clusters on Marconi. Despite BDW performing faster, we see some considerably good advantages on KNL brought about by the high-bandwidth MCDRAM operating as a large L3 cache. Because of this, elapsed times on KNL do not increase so much as in BDW when hitting the communication bottleneck described earlier. KNL also consistently shows better parallel efficiency than BDW and even exhibits superscaling attributed to cache effects from MCDRAM.

CHAPTER 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we have presented our massively parallel implementation of the density matrix renormalization group (DMRG) algorithm for studying the ground state properties of strongly correlated systems. In writing the application, we have used the PETSc library to systematically handle parallel linear algebra objects and operations, and the SLEPc library for the efficient diagonalization of large sparse Hamiltonian matrices, both parallelized in a distributed memory manner through Message Passing Interface (MPI). We have implemented our code for the one-dimensional spin-1/2 Heisenberg chain, a known model which we use to check for accuracy.

We measured the performance and scalability of our implementation on the Broadwell (BDW) and Knights Landing (KNL) sections of Marconi, a class Tier-0 supercomputer. Our performance results demonstrate good scalability up to the order of one thousand cores. We identified limitations in scalability such as communication bottlenecks, fitting the problem size in memory, and saturation of the memory bandwidth. We also made performance comparisons between BDW and KNL and we found that the latter exhibits better parallel efficiency due to its high bandwidth memory.

We have performed the numerical simulations on cases with large number of kept states m that are often intractable on single-node computational systems. This thesis demonstrates the framework for implementing the full DMRG calculations for two-dimensional systems that require large number of m which was made possible by treating the problem with an HPC approach.

5.2 Future Work

This implementation serves as the initial stage for an ongoing Italian SuperComputing Resource Allocation Class C (ISCRA C) project which aims to study topological phases in two-dimensional systems. The next step for this work is implementing the full two-dimensional (2D) finite DMRG algorithm. This will involve the same steps in the workflow presented in this thesis for infinite DMRG but with a greater memory consumption since all operators have to be stored after after each iteration, and with more interacting sites and more terms in the Hamiltonian.

Our current approach relies on creating the large superblock Hamiltonian matrix explictly in memory thus greatly limiting the number of states m that can be explored for a given number of computational nodes. Alternatively, instead of creating the large sparse matrix, we may also calculate the matrix vector product of the Hamiltonian which is the only operation required by iterative solvers like Krylov-Schur. This matrix-free approach can be implemented in PETSc through the `MATSHELL` interface. However, this will involve restructuring our Kronecker product implementation, and this may entail greater cost in computing time since more operations are involved during the matrix vector product.

Additionally, our tests on the two sections of Marconi has shown that KNL has superior parallel efficiency than BDW due to the presence of MCDRAM. In our simulations, however, the MCDRAM operates in cache modes making it invisible to our application. This advantage may further be exploited by using MCDRAM in flat mode as an addressable lower-capacity piece of memory, and manually controlling the partitioning of data between MCDRAM and DDR4 using the `memkind` library. [38]

Bibliography

[1] Patrick a. Lee, Naoto Nagaosa, and Xiao-Gang Wen. “Doping a Mott insulator: Physics of high-temperature superconductivity”. In: *Rev. Mod. Phys.* 78.1 (2006), pp. 17–85. ISSN: 0034-6861. DOI: 10.1103/RevModPhys.78.17. URL: http://link.aps.org/doi/10.1103/RevModPhys.78.17.

[2] C. Lacroix, P. Mendels, and F. Mila, eds. *Introduction to Frustrated Magnetism*. Springer Series in Solid-State Sciences Vol. 164, 2010.

[3] H C Jiang, Z Y Weng, and D N Sheng. “Density Matrix Renormalization Group Numerical Study of the Kagome Antiferromagnet”. In: *Physical Review Letters* 101.11 (Sept. 2008), pp. 117203–4.

[4] E Gibney and D Castelvecchi. *Physics of 2D exotic matter wins Nobel.* Vol. 538. Nature, 2016.

[5] Steven R White. “Density matrix formulation for quantum renormalization groups”. In: *Physical Review Letters* 69.19 (1992), pp. 2863–2866.

[6] Susumu Yamada, Masahiko Okumura, and Masahiko Machida. “Direct Extension of Density-Matrix Renormalization Group to Two-Dimensional Quantum Lattice Systems: Studies of Parallel Algorithm, Accuracy, and Performance”. In: *Journal of the Physical Society of Japan* 78.9 (Sept. 2009), pp. 094004–5.

[7] E M Stoudenmire and Steven R White. “Studying Two-Dimensional Systems with the Density Matrix Renormalization Group”. In: *Annual Review of Condensed Matter Physics* 3.1 (Mar. 2012), pp. 111–128.

[8] F B Ramos and J C Xavier. “N-leg spin-S Heisenberg ladders: A density-matrix renormalization group study”. In: *Physical Review B* 89.9 (Mar. 2014), pp. 094424–7.

[9] Mischa Thesberg and Erik S Sørensen. “An Exact Diagonalization Study of the Anisotropic Triangular Lattice Heisenberg Model Using Twisted Boundary Conditions”. In: *arXiv.org* 1 (June 2014), p. 115117. arXiv: 1406.4083v2 [cond-mat.str-el].

[10] M Troyer and U J Wiese. “Computational complexity and fundamental limitations to fermionic quantum Monte Carlo simulations”. In: *Physical Review Letters* 95.12 (2005).

[11] Adrian E Feiguin. “The Density Matrix Renormalization Group”. In: *Strongly Correlated Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, Apr. 2013, pp. 31–65.

[12] Richard P Feynman. “Simulating physics with computers”. In: *Int. J. Theor. Phys.* 21.6-7 (1982), pp. 467–488.

[13] H Fehske, R Schneider, and A Weisse. *Computational many-particle physics*. Ed. by H Fehske, R Schneider, and A Weisse. Vol. 739. Lecture Notes in Physics. Berlin: Springer, 2008.

[14] Anders W Sandvik and Juhani Kurkijärvi. “Quantum Monte Carlo simulation method for spin systems”. In: *Physical Review B (Condensed Matter)* 43.7 (Mar. 1991), pp. 5950–5961.

[15] U Schollwöck. “The density-matrix renormalization group”. In: *Reviews of Modern Physics* 77.1 (Jan. 2005), pp. 259–315.

[16] Kenneth G Wilson. “The Renormalization Group: Critical Phenomena and the Kondo Problem”. In: *Rev. Mod. Phys.* 47.4 (1975), pp. 773–840.

[17] R J Furnstahl and K Hebeler. “New applications of renormalization group methods in nuclear physics”. In: *Reports on Progress in Physics* 76.12 (Nov. 2013), pp. 126301–26.

[18] Ö Legeza et al. “Advanced density matrix renormalization group method for nuclear structure calculations”. In: *Physical Review C* 92.5 (Nov. 2015), pp. 051303–5.

[19] Steven R White and Richard L Martin. “Ab initio quantum chemistry using the density matrix renormalization group”. In: *The Journal of Chemical Physics* 110.9 (Mar. 1999), pp. 4127–4130.

[20] Garnet Kin-Lic Chan and Sandeep Sharma. “The Density Matrix Renormalization Group in Quantum Chemistry”. In: *Annual Review of Physical Chemistry* 62.1 (May 2011), pp. 465–481.

[21] Gabriele De Chiara et al. “Density Matrix Renormalization Group for Dummies”. In: *Journal of Computational and Theoretical Nanoscience* 5.7 (July 2008), pp. 1277–1288.

[22] S Yan, D A Huse, and S R White. “Spin-Liquid Ground State of the S = 1/2 Kagome Heisenberg Antiferromagnet”. In: *Science* 332.6034 (2011), pp. 1173–1176.

[23] Satish Balay et al. *PETSc Users Manual*. Tech. rep. ANL-95/11 - Revision 3.7. Argonne National Laboratory, 2016.

[24] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. “SLEPc: A Scalable and Flexible Toolkit for the Solution of Eigenvalue Problems”. In: *ACM Trans. Math. Software* 31.3 (2005,), pp. 351–362. DOI: http://dx.doi.org/10.1145/1089014.1089019.

[25] Samuel Williams et al. “Optimization of sparse matrixvector multiplication on emerging multicore platforms”. In: *Parallel Computing* 35.3 (Mar. 2009), pp. 178–194.

[26] Marlon E Brenes-Navarro. *Parallel implementation of the Krylov subspace techniques for unitary time evolution of disordered quantum strongly correlated systems*. Ed. by Antonello Scardicchio, Ivan Girotto, and Vipin Varma. Scuola Internazionale Superiore di Studi Avanzati, Dec. 2016.

[27] Marlon Brenes et al. “Massively parallel implementation and approaches to simulate quantum dynamics using Krylov subspace techniques”. In: *arXiv.org* (Apr. 2017). arXiv: 1704.02770v1 [physics.comp-ph].

[28] Adrian E Feiguin. “The Density Matrix Renormalization Group and its time-dependent variants”. In: *Lectures on the Physics of Strongly Correlated Systems XV: Fifteenth Training Course in the Physics of Strongly Correlated Systems. AIP Conference Proceedings*. Department of Physics and Astronomy, University of Wyoming, Wyoming, USA 82071. AIP, Dec. 2011, pp. 5–92.

[29] Satish Balay et al. *PETSc Web page*. http://www.mcs.anl.gov/petsc. 2017. URL: http://www.mcs.anl.gov/petsc.

[30] V Hernandez, J E Roman, and A Tomas. “A parallel Krylov-Schur implementation for large Hermitian and non-Hermitian eigenproblems”. In: *PAMM* 7.1 (Dec. 2007), pp. 2020083–2020084.

[31] V. Hernandez et al. *Krylov-Schur Methods in SLEPc*. Tech. rep. STR-7. Available at http://slepc.upv.es. Universitat Politècnica de València, 2007.

[32] J. E. Roman et al. *SLEPc Users Manual*. Tech. rep. DSIC-II/24/02 - Revision 3.7. D. Sistemes Informàtics i Computació, Universitat Politècnica de València, 2016.

[33] Jonas Maziero. "Computing partial traces and reduced density matrices". In: *International Journal of Modern Physics C* 28.01 (Jan. 2017), pp. 1750005–17.

[34] James R. Garrison and Ryan V. Mishmash. *Simple DMRG*. Version v1.0.0. DOI: 10.5281/zenodo.1068359. URL: https://github.com/simple-dmrg/simple-dmrg.

[35] CINECA. *MARCONI User Guide*. accessed 24 November 2017. 2017. URL: https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%5C%3A+MARCONI+UserGuide.

[36] Beat Frischmuth, Beat Ammon, and Matthias Troyer. "Susceptibility and low-temperature thermodynamics of spin-1/2 Heisenberg ladders". In: *Physical Review B* 54.6 (Aug. 1996), R3714–R3717.

[37] Vipin Kumar et al. *Introduction to Parallel Computing*. 1994.

[38] Intel. *Memkind*. Version v1.6.0. URL: http://memkind.github.io/memkind/.