Master in High Performance Computing

Massively Parallel Approaches to Frustrated Quantum Magnets

Supervisors:
Ivan Girotto,
Marcello Dalmonte

Candidate:
Alejandra María Foggia

4th edition
2017–2018

Contents

Chapter 1

Introduction

Quantum magnets are one of the paradigmatic platforms for the investigation of strongly correlated, highly entangled states of matter. On the experimental side, rapid advances in the field of magnetic materials and cold atoms in optical lattices are nowadays heavily challenging theoretical methods. In parallel, these impressive experimental capabilities are providing novel probing tools that might be used to detect and understand entangled phases without direct access to the system wave function.

A key question is thus if, and to which extent, it is possible to utilize these probing tools to theoretically understand highly entangled phases of matter from a perspective based on experimentally available probes. A particularly promising route is the exploration of dynamical structure factors that are experimentally accessible, for instance, in inelastic neutron scattering experiments.

An application to simulate the dynamical structure factor with Dynamical Lanczos method is presented in this thesis. The physical system is modeled by 1/2-spins placed in a one-dimensional chain or a two-dimensional square lattice with periodic boundary conditions and with energies governed by a XXZ antiferromagnetic Hamiltonian. No translational invariance is imposed, as disordered systems want to be studied, but conservation of the total magnetization is a restriction enforced to the system.

The characteristics of the problem require the use of Exact Diagonalization (ED) technique to search for the eigenstates of the system, approximations are ruled out. This method demands the construction, and usually the storage, of the basis elements and the Hamiltonian matrix, which requires a large amount of memory. This amount increases exponentially with the number of spins present in the system, reaching already tens of terabytes for 38 spins. Moreover, this has a direct effect on the amount of computational resources needed to obtain the eigenstates of the system.

The reasons mentioned before demand a High Performance Computing (HPC) approach if the goal is to be as realistic as possible. This is the purpose of this project, and all the efforts were put in making it possible.

Chapter 2

Background

2.1 Frustrated Magnets

There exist magnetic materials in which it is not possible to satisfy all the interactions between every pair of spins (localized magnetic moments). In other words, the minimum of the system energy does not correspond with the minimum energy of each bond. This property of the materials is called *frustration* and it can have its origin in different situations, those being competing exchange interactions, particular lattice structure or anisotropy in the material. There are a few clear and simple examples of this phenomenon. When it comes to geometrical frustration, the two-dimensional (2D) triangular lattice with antiferromagnetic interactions (Figure 2.1) is the paradigmatic example where the effect is immediately seen: if two of the three bonds satisfy their interaction, meaning, for example, that two spins point upward while the third one points downward, then the last bond is compelled to be unsatisfied. Likewise, a 2D square lattice with nearest neighbours interaction, J_1, and next nearest neighbours interaction, J_2, both antiferromagnetic, shows frustration [1].

One of the characteristic features of frustrated magnets, which is highly unusual in quantum systems, is the possibility that long range magnetic order (LRO) does not take place even down to zero temperatures. Classical (thermal) and, more importantly, quantum fluctuations play an important role in this matter. In models

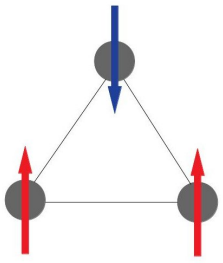


Figure 2.1: Frustrated two-dimensional antiferromagnetic triangular lattice: top spin (blue) satisfies the interaction with both red spins but these do not satisfy their mutual interaction.

described by large S spin variable, thermal fluctuations govern the dynamics at high temperatures and they cease to exist as temperature lowers allowing the system to freeze or order. Instead, for small spin numbers, e.g. 1/2-spins, quantum fluctuations have a significant effect and are the ones that suppress LRO. When frustration is taken to the extreme it's possible to get a *quantum spin liquid* (QSL) ground state. Contributing to the characterization of both spin liquid phases and one-dimensional (1D) frustrated magnets is the leading purpose behind the work carried out in this thesis.

The endeavor to characterize QSL state of matter started long ago and even now a complete classification is an open problem, a 'positive' description is often challenging due to their subtle nature. QSLs are more famous for being depicted for what they do not do: order even a very low or zero temperatures, rather than being characterized for what they actually are. It can be said confidently that, in QSLs, spins present high entanglement between widely separated lattice sites, and this allows such states to show an "exotic phenomenon": they support non-local excitations with fractional quantum numbers named *spinons*1. One of the motivations of some scientists to study this kind of frustrated systems, specially the ground state, lays in the conviction that they may be associated with, also exotic, forms of superconductivity.

On the numerical simulation level, many approaches exist to study these systems. Quantum Monte Carlo (QMC) methods provide ways of simulating large number of particles [2,3] with less computational complexity than other methods, but it is not suited for fermions (due to the "sign problem") neither for the vast majority of frustrated spin systems. Exact diagonalization (ED) approach does not have this "sign problem" but the computational effort is such that the number of particles simulated has to be drastically reduced compared to QMC in order to be able to fit the problem in the current machines available and to obtain a result over realistic timescales. One last method has been recently introduced, Density Matrix Renormalization Group (DMRG) [4]. It does not suffer from the "sign problem" either and it allows the study of a larger number of lattice sites than ED, by finding an appropriate truncation of the Hilbert space. Even if this technique is well suited for 1D systems, dynamical properties, such as the dynamical structure factor, are harder to obtain with this method than ground state energies. This limitation of DMRG is one of the reasons why we choose to use ED in our work; importantly, the problems we plan to tackle are potentially very sensitive to computational errors, since they rely on knowledge of the spectral properties. As such, any truncation at the Hilbert space level is expected to be detrimental in a rather uncontrolled fashion.

As always happens in physics, there exists the strong willingness for theoretical research (which includes simulations) to be validated by experimental results. However, identifying QSLs in experiments is not an easy task, since there is no unique physical quantity that can be used to characterize a frustrated magnet. Consequently, many quantities are studied and several techniques are used [1]. Among experimental techniques, one of the most relevant ones is inelastic neutron-scattering (see Ref. [5] and references therein): experiments provide information

about the spin correlation functions, and these are also related to the presence of order in the sample. There is also nuclear magnetic resonance and muon spin resonance, which are used to study the presence or absence of static moments in material in the ground state. Other quantities of interest are magnetic susceptibility: it is possible to observe the suppression of the magnetic order by measuring this quantity as a function of temperature; and heat capacity, which is related to the degrees of freedom of the material, and through this one, one can study the density of states.

2.1.1 Inelastic Neutron Scattering

The basics of this technique include an incident neutron of known wavevector $\mathbf{k}_i$ that is scattered by the sample into an outgoing neutron wavevector $\mathbf{k}_f$. In the experiment, it is possible to measure the change in the direction and the magnitude of the neutron wavevector. These quantities are related to the momentum and energy of the neutron by

$$\begin{aligned} \Delta\mathbf{p} &= \hbar\Delta\mathbf{k} \\ \Delta E &= \hbar\omega. \end{aligned} \tag{2.1}$$

As momentum and energy are conserved quantities in the experiment, the variation of these in the neutron are exactly mirrored by the ones in the sample.

The scattering intensity is proportional to the time and spatial Fourier transform of the two-spin correlation function of the sample

$$S^{\gamma}(\mathbf{k},\omega) = \sum_{\mathbf{r},t} \mathrm{e}^{i(\omega t+\mathbf{r}\cdot\mathbf{k})}\langle S^{\gamma}_{\mathbf{r}}(t)S^{\gamma}_{0}(0)\rangle, \tag{2.2}$$

where S^{γ} is a spin operator (like S^z, S^+, S^-, etc.). This quantity, called *dynamical structure factor* (DSF), is the one we are interested in and the one we are simulating in this work. It can also be rewritten as

$$S^{\gamma}(\mathbf{q},\omega) = \sum_{n} |\langle\psi^X_n|S^{\gamma}_{\mathbf{q}}|\Psi_0\rangle|^2\delta(\omega-(E_n-E_0)), \tag{2.3}$$

where $S^{\gamma}_{\mathbf{q}} = \frac{1}{\sqrt{L}}\sum_{i=1}^{L}\mathrm{e}^{i\mathbf{q}\cdot\mathbf{r}_i}S^{\gamma}_{\mathbf{r}_i}$, with $\mathbf{q} = \frac{2\pi}{L}\mathbf{k}, |k| = 0,1,\ldots,L-1$, and L is the number of lattice sites. In this equation, Ψ_0 is the ground state of the Hamiltonian, while $\{\psi^X_n\}$ are some of the excited states of the Hamiltonian subspace that corresponds to the application of S^{γ} to the basis states of the $S^z_{total} = 0$ subspace (where the ground state energy is found). In the context of spin liquids, a recent experiment has been performed along the lines described here [5]. In the context of one-dimensional Heisenberg models, see, e.g., Ref. [6].

2.1.2 Exact Diagonalization

The Exact Diagonalization numerical technique gets its name from the fact that one exactly solves the time-independent Schrödinger equation $\mathcal{H}|\psi\rangle = E|\psi\rangle$,

thus this procedure involves the numerical diagonalization of the Hamiltonian matrix without approximations. This method can be applied in many classes of spin systems and it allows one to compute the expectation value of almost any observable. As one can anticipate, it requires the creation, and sometime storage, of the basis elements and the matrix, which implies a big memory consumption when going above 30 spins since its Hilbert space is of the order of 10^9 elements.

An usual ED program consists in four stages [1]:

1. Creation of the basis elements in the Hilbert space of the Hamiltonian.
2. Creation of the Hamiltonian.
3. Diagonalization.
4. Computation of the observable desired.

In the next chapter, we will go over each of these steps giving detailed information about its implementation, but for now we can discuss briefly and generally each part.

During the creation of the basis states, usually some symmetries of the problem are applied, like charge conservation, total magnetization conservation or momentum conservation (translational symmetry), just to mention a few. The size of the final Hilbert space and the complexity of the elements' creation depends on how many of those are applied to the problem. In the problem treated here, only total magnetization conservation is used and this reduces the space from $1073741824 \sim 10^9$ elements to $155117520 \sim 10^8$ elements, for 30 spins, since the ground state of the system is in the $S^z_{total} = 0$ subspace. In this part one usually has to decide the distribution of the basis elements across the memory of multiple nodes.

Coming to the Hamiltonian construction, the computational effort and the coding difficulty of this part depends exclusively on the operands that are involved. Specifically in our problem, the Hamiltonian is the antiferromagnetic XXZ model

$$\mathcal{H} = -|J| \sum_{\langle i,j \rangle} \{ \frac{1}{2} (S^+_i S^-_j + S^-_i S^+_j) + \Delta S^z_i S^z_j \}. \tag{2.4}$$

The third part, the diagonalization, is the one in which most of the computing effort is done and, in our case, is handled by two external libraries, PETSc and SLEPc, specialized and optimized to solve this kind of problems. Many solvers can be used; in general, bibliography and articles on this topic refer to the Lanczos method as the usual way to proceed with the diagonalization. Based on benchmarks of the code, and we chose instead to use a Krylov-Schur method to obtain the ground state energy.

2.2 Lanczos and Krylov-Schur Methods

The list of numerical methods available to solve eigenvalue problems is long. It includes, among others, the *Power method*, a single vector iteration method;

Subspace Iteration methods, that can be viewed as a generalization of the Power method; *projection methods* like the *Rayleigh-Ritz procedure*, and many *Krylov Subspace methods*, that are based on the Rayleigh-Ritz procedure. This last class of methods includes the *Lanczos algorithm* (usually, for Hermitian matrices) and *Arnoldi* and *Krylov-Schur methods* (for non Hermitian problems), all of them better used to compute eigenvalues at the extreme of the spectrum. For interior eigenvalues, there exists *Davidson method*, which is another projection method used in conjunction with *preconditioners* [7].

The problem treated in this thesis anticipates the need of external eigenvalues, more precisely the ones closest to the ground state of the system, therefore Lanczos and Krylov-Schur algorithms are suitable choices. As both of them are based on the Rayleigh-Ritz procedure, it is beneficial to start by having a look at it [7]. Suppose a matrix $A \in \mathbb{C}^{n\times n}$ and a subspace of $\mathbb{C}^n$ of dimension $m \leq n$, $\mathcal{K}$. The eigenvalue problem consists in finding the pair $\{\lambda, u\}$ such that

$$Au = \lambda u. \tag{2.5}$$

The idea now is to find an approximate pair $\{\tilde{\lambda}, \tilde{u}\}$, called *Ritz value* and *Ritz vector*, $\tilde{\lambda} \in \mathbb{C}$ and $\tilde{u} \in \mathcal{K}$, such that

$$(A\tilde{u} - \tilde{\lambda}\tilde{u}, v) = 0, \quad \forall\ v \in \mathcal{K}. \tag{2.6}$$

Suppose say that $\{v_1, v_2, \ldots, v_m\}$ is an orthonormal basis of $\mathcal{K}$ and V is a matrix whose columns are the vectors $v_i \in \mathbb{C}^n$. It is possible to solve the approximate problem numerically by writing it into the new basis

$$\tilde{u} = Vy; \tag{2.7}$$

hence the new problem is given by

$$(AVy - \tilde{\lambda}Vy, v_j) = 0,\ j = 1, \ldots, m. \tag{2.8}$$

This implies that y and $\tilde{\lambda}$ have to satisfy $By = \tilde{\lambda}y$, where $B = V^H AV$ is a projection of A onto $\mathcal{K}$. With this procedure one can find the eigenpairs of A by solving a smaller matrix ($m \times m$). How much smaller this matrix is depends on the problem, the part of the spectrum one is inspecting and the number of eigenpairs to find.

Both Lanczos and Krylov-Schur algorithms construct, iteratively, the subspace $\mathcal{K}$ starting from the matrix A and a vector v_1. This space has the particular "shape"

$$\mathcal{K} = \text{span}\{v_1, Av_1, A^2v_1, \ldots, A^{m-1}v_1\}, \tag{2.9}$$

and it is called *Krylov* subspace. The main difference between the two methods is the characteristics of the projected matrix A onto $\mathcal{K}$: in the first case, it is a tridiagonal matrix, while in the second one, it assumes a real Schur form, that means a quasi-triangular form displaying eigenvalues in the 1×1 and 2×2 diagonal blocks [8].

2.2.1 Lanczos algorithm

With Lanczos method one can reduce the matrix A to a tridiagonal form with the following recurrence formula

$$\beta_{j+1}v_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}, \tag{2.10}$$

where α_i are the values of the diagonal and β_i are the values in the first sub- and supradiagonals (it is symmetric). This procedure will generate two $n \times n$ matrices, one tridiagonal, T, and V, such that

$$AV - VT = 0 \quad \Rightarrow \quad V^H AV = T; \tag{2.11}$$

but the goal is to end up dealing with a much smaller one. When the procedure is stopped after m steps, one ends up with m orthogonal vectors $\{v_i\}_{i=1}^m$, called *Lanczos vectors*, such that when arranged as columns of V_m, the *Lanczos decomposition* is obtained

$$AV_m - V_m T_m = \beta_{m+1} v_{m+1} e_m^*. \tag{2.12}$$

Because v_{m+1} is orthogonal to V_m by construction, one obtains, similarly to 2.11, that $V_m^H AV_m = T_m$. This matrix is the orthogonal projection of A onto the Krylov subspace, which implies that it is possible to compute Rayleigh-Ritz approximations to the eigenpairs of A and the matrix that has to be used, T, is easier to diagonalize, computational-wise, than the original one. In other words, it is possible to get the eigenpairs $\{\lambda_i, y_i\}$ of T_m and obtain the approximate eigenpairs of A, $\{\lambda_i, u_i\}$ as $u_i = V_m y_i$ (the subscripts in T and V indicate the dimension of the space).

Lanczos method seems good and efficient but there is an important fault in it: as the Lanczos vectors lose mutual orthogonality, multiple copies of the already converged Ritz values appear, and the process gives wrong Ritz values as converged [9]. The straightforward answer to this problem is orthogonalize the Lanczos vectors after every iteration or after some iterations have been made; but the procedure, apart from involving more calculations, it also involves a higher memory consumption as all Lanczos vectors need to be kept in memory. As a solution to this, a few restarted algorithms have been developed. The idea behind this is to reduce the storage requirement by performing multiple m-step Lanczos factorizations using better initial vectors each time. This is the approach used in this thesis when Lanczos algorithm is mentioned, and it is called *explicitly restated Lanczos*.

2.2.2 Krylov-Schur Algorithm

In the same way it is possible to obtain the Lanczos decomposition (2.12), it is also feasible to get a *Krylov-Schur decomposition* that has the form

$$AV_m - V_m S_m = v_{m+1} b_{m+1}^*, \tag{2.13}$$

where the difference, as mentioned before, lies in the shape of the matrix S_m, that is the counterpart of V_m in Equation 2.12.

The Krylov-Schur method is equivalent to an Arnoldi decomposition (in which case the matrix in question is upper Hessenberg). One step of the method consists of an expansion phase followed by a contraction phase. During the former one, the Krylov sequence is extended, while in the latter, the unwanted Ritz values are removed from the decomposition [10].

The expansion phase starts with the Arnoldi algorithm [1]. This method, shown in Algorithm 2.1, consists in obtaining iteratively the vectors v_i of the subspace and the upper Hessenberg matrix H_m.

Algorithm 2.1: Arnoldi's algorithm.

```
input: matrix A, int m (steps), initial vector v_1
output: matrix V_m, H_m

for j ∈ [1,m)
    h_{i,j} = (A v_j, v_i), i ∈ [1,j]
    w_j = A v_j - Σ_{i=1}^{j} h_{i,j} v_i
    h_{j+1,j} = || w_j ||_2 , if h_{j+1,j} = 0 stop
    v_{j+1} = w_j / h_{j+1,j}
```

After this, the QR algorithm (for more details refer to [7] and [10]) is applied to S_m such that at $T_m = Q^* S_m Q$ has real Schur form, and at this point the Ritz values are available from the diagonal blocks of T_m.

The contraction phase starts by moving the unwanted Ritz values to the lower right part of T_m. This is accomplished by means of orthogonal transformations. Later, a truncation is executed by just removing the columns of V_m and T_m that correspond to the unwanted Ritz values.

When the matrix A is symmetric, the Krylov-Schur method is equivalent to the thick-restarted Lanczos procedure.

2.3 Dynamical Lanczos Method

As mentioned before, the quantity of interest in this thesis is the Dynamical Structure Factor (Equation 2.3). It is possible to compute it by the, so called, *Dynamical Lanczos method*. The method commonly used for diagonalization is Lanczos, but it can be replaced by the Krylov-Schur method. The procedure goes as follows:

1. Compute the ground state of the system, $|\Psi_0\rangle$.

2. Once the desired spin operator is chosen, $S^{\gamma}_{\vec{q}}$, create the initial vector of a

[1] It is worth mentioning that the Lanczos algorithm presents a very similar form (for a more thorough description refer to [9]).

new Lanczos procedure (marked with X superscript) as

$$v_1^X = \frac{S_{\vec{q}}^{\gamma}|\Psi_0\rangle}{\langle\Psi_0|S_{\vec{q}}^{\gamma\dagger}S_{\vec{q}}^{\gamma}|\Psi_0\rangle}. \tag{2.14}$$

3. Perform an m-step Lanczos procedure with v_1^X as the initial vector. The matrix to diagonalize depends on the selected operator: if it is S^z, then it is the same Hamiltonian used for the ground state; if it is S^+ or S^-, then a new basis and Hamiltonian have to be constructed, in which the total magnetization is either 1 or -1, respectively.

Another equivalent expression for Equation 2.3 can be obtained after some algebra, having in mind that the eigenvectors, $\{\psi_k^X\}_{k=1}^m$, of the tridiagonal matrix of a Lanczos procedure can be written as a linear combination of the Lanczos vectors, $\{v_i\}_{i=1}^m$, obtained from that same procedure,

$$\psi_k^X = \sum_{i=1}^{m}(c_k)_i v_i. \tag{2.15}$$

With this and with Equation 2.14, it is possible to write 2.3 as follows:

$$\begin{aligned}
S^{\gamma}(\mathbf{q},\omega) &= \sum_n |\langle\psi_n^X|S_{\mathbf{q}}^{\gamma}|\Psi_0\rangle|^2\delta(\omega-(E_n^X-E_0)) \\
&= \sum_n \left[\left(\sum_i (c_n)_i v_i^X\right)\left(\langle\Psi_0|S_{\vec{q}}^{\gamma\dagger}S_{\vec{q}}^{\gamma}|\Psi_0\rangle v_1^X\right)\right]^2 \delta(\omega-(E_n^X-E_0)) \qquad (2.16)\\
&= \sum_n \left[\sum_i (c_n)_i v_i^X \langle\Psi_0|S_{\vec{q}}^{\gamma\dagger}S_{\vec{q}}^{\gamma}|\Psi_0\rangle v_1^X\right]^2 \delta(\omega-(E_n^X-E_0)) \qquad (2.17)\\
&= \sum_n \left[(c_n)_1\langle\Psi_0|S_{\vec{q}}^{\gamma\dagger}S_{\vec{q}}^{\gamma}|\Psi_0\rangle\right]^2 \delta(\omega-(E_n^X-E_0)) \qquad (2.18)
\end{aligned}$$

4. Obtain "a few" of the eigenpairs $\{\psi_i, E_i^X\}_{i=1}^m$ and keep only the first coordinate of them and the energy.

2.4 PETSc/SLEPc

The method chosen to study frustrated magnetic systems, Dynamical Lanczos, requires the diagonalization of the Hamiltonian matrix several times to get the dynamical structure factor. Moreover, if one is willing to study the effect of disorder, this single, already computational demanding, procedure is repeated multiple times for different values of the "disorder" constant Δ in Equation 2.4.

The challenge here arises when the matrix's linear dimension increases up to a few thousand millions, and even thought it is a sparse matrix, the diagonalization becomes a computational issue. On top of this, dealing with that amount of

data requires a distributed memory approach, that implies, not only, allocating the memory in the correct way, but also, parallelizing the algorithm one is using, and doing all that efficiently. For these reasons, we have chosen to rely on two open source libraries designed to deal with these kind of problems (and many more): PETSc and SLEPc. These libraries have already been used in previous thesis [11, 12] and they have demonstrated to be reliable and to provide good performance.

PETSc, that stands for *Portable, Extensive Toolkit for Scientific Computation*, as from its home page, is "a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations" [13]. The software supports a few levels of parallelization: Message Passing Interface (MPI), GPUs (with CUDA or OpenCL), and the hybrid version of the previous two. Apart from these, it's feasible to add OpenMP to improve the scalability. PETSc provides objects to deal with simple data types (integers, doubles, complex numbers) and with more complex ones: vectors (`Vec`) and matrices (`Mat`). Many functions allow one to create, allocate, manipulate, communicate and destroy all these objects, in a sequential way or in a parallel one. Particularly, in the latter case, these functions permit the user to do everything without being concerned about the details of the allocation or the communication of the distributed objects. A rich set of preconditioners and linear and non linear solvers is provided, among many other functionalities to work with domains. The software interfaces with standard de facto libraries like LAPACK, BLAS and FFTW, which gives the user freedom to link it against other powerful libraries, e.g. MKL, and renders it very versatile and useful in many scenarios.

SLEPc is a software built on top of PETSc. It's the *Scalable Library for Eigenvalue Problem Computations*, and it provides a platform for "large scale sparse eigenvalue problems on parallel computers" [14]. This software offers efficient parallel implementations of the well-known algorithms, in particular for Lanczos and Krylov-Schur, making use of the data structures and functions already available in PETSc.

Chapter 3

Implementation

3.1 Overview of the Code

The code is written in C++ language, in an object oriented fashion. It is composed by building blocks that mirror the organization of the underlying mathematical description of the physical problem: the basis of the system, the lattice (distribution of spins), the Hamiltonian, the solver (diagonalization of the matrix) and the operator $S^{\gamma}_{\vec{q}}$. Each of these blocks is represented by a C++ class and they are all related, in the sense that some are needed for the construction of others.

The code has not reached its final stage yet, it's still on production, however there's a clear idea of what the final picture should look like. As explained in Section 2.3, the work flow should be like it's shown in Figure 3.1; each section below has a reference to a node of the flowchart in said Figure.

Initialization

(A) The system size (number of spins), the lattice configuration and the type of spins operator used for the correlation function are input parameters. Immediately, an `Environment` object is instantiated. This object takes care of correctly initiating and finalizing the SLEPc environment, which handles the initialization and finalization of the PETSc environment, which, in turn, performs the MPI calls to set the parallel environment. This `Environment` object is used in the construction of all the other classes, as it standardizes the communicator for every building block.

Lattice

(B) The ground state of the system and the correlation functions depend on the relative position of the spins. The `Lattice` class reflects this: it creates a list of the neighbours of each spin depending on the configuration. This list takes into account the *periodic boundary conditions* used in the problem treated here. One can choose between a `1D chain` lattice, a `2D square` lattice or a `2D honeycomb` lattice, and it is a input parameter. This list is a `std::vector<PetscInt>` and it is replicated in the memory of every MPI process. This does not create a problem

because the number of elements in the vector is two or three (depending on the lattice) times larger that the total number of spins, which is around 38.

System Basis

(C) Another building block of the mathematical description of the problem, and also the first step in the Exact Diagonalization procedure (Section 2.1.2), is the basis of the system, represented by the `Basis` class. The purpose of this class is to

- compute the number of spins up and spins down given the total number of spins (`nspins`) and the total magnetization (`total_mag`), which is conserved;
- construct and hold the vector with the basis elements of the system `std::vector<PetscInt>`;
- construct and hold the vector with the basis elements in bit representation `std::vector<boost::dynamic_bitset<>>`, that is used during developing and testing stages only.

An important issue in this part is the size of vector containing the basis elements. If one considers the full Hilbert space of the problem then one has to deal with $\mathsf{N}_{\mathcal{H}_{\mathsf{full}}} = 2^{\mathtt{nspins}}$ elements. However, as the problem setting presented in this work imposes total magnetization conservation, the Hilbert space is reduced and the number of elements is given by

$$\mathsf{N}_{\mathcal{H}_{\mathsf{red}}} = \frac{\mathtt{nspins}!}{N_\uparrow! N_\downarrow!}, \tag{3.1}$$

where $N_\uparrow$ and $N_\downarrow$ are the number of spins pointing up and down, correspondingly. In Table 3.1, the size of both Hilbert spaces are shown, for the case of $S^z_{total} = 0$. It is clear that there is an order of magnitude of difference between both spaces size. It is important to keep in mind that all these elements are going to be constructed

nspins	$\mathsf{N}_{\mathcal{H}_{\mathsf{full}}}$	$\mathsf{N}_{\mathcal{H}_{\mathsf{red}}}$
18	262,144	48,620
22	4, 194, 304	705, 432
26	67, 108, 864	10, 400, 600
30	1, 073, 741, 824	155, 117, 520
32	4, 294, 967, 296	601, 080, 390
34	17, 179, 869, 184	2, 333, 606, 220
36	68, 719, 476, 736	9, 075, 135, 300
38	274, 877, 906, 944	35, 345, 263, 800

Table 3.1: Sizes of the full and reduced Hilbert spaces, considering the total magnetization conservation, for some number of spins.

and, possibly [1], stored in the machine's memory. This is already $\sim 72\,\mathrm{GB}$ of

[1] There is another approach in which the matrix is not stored in memory but constructed as needed.

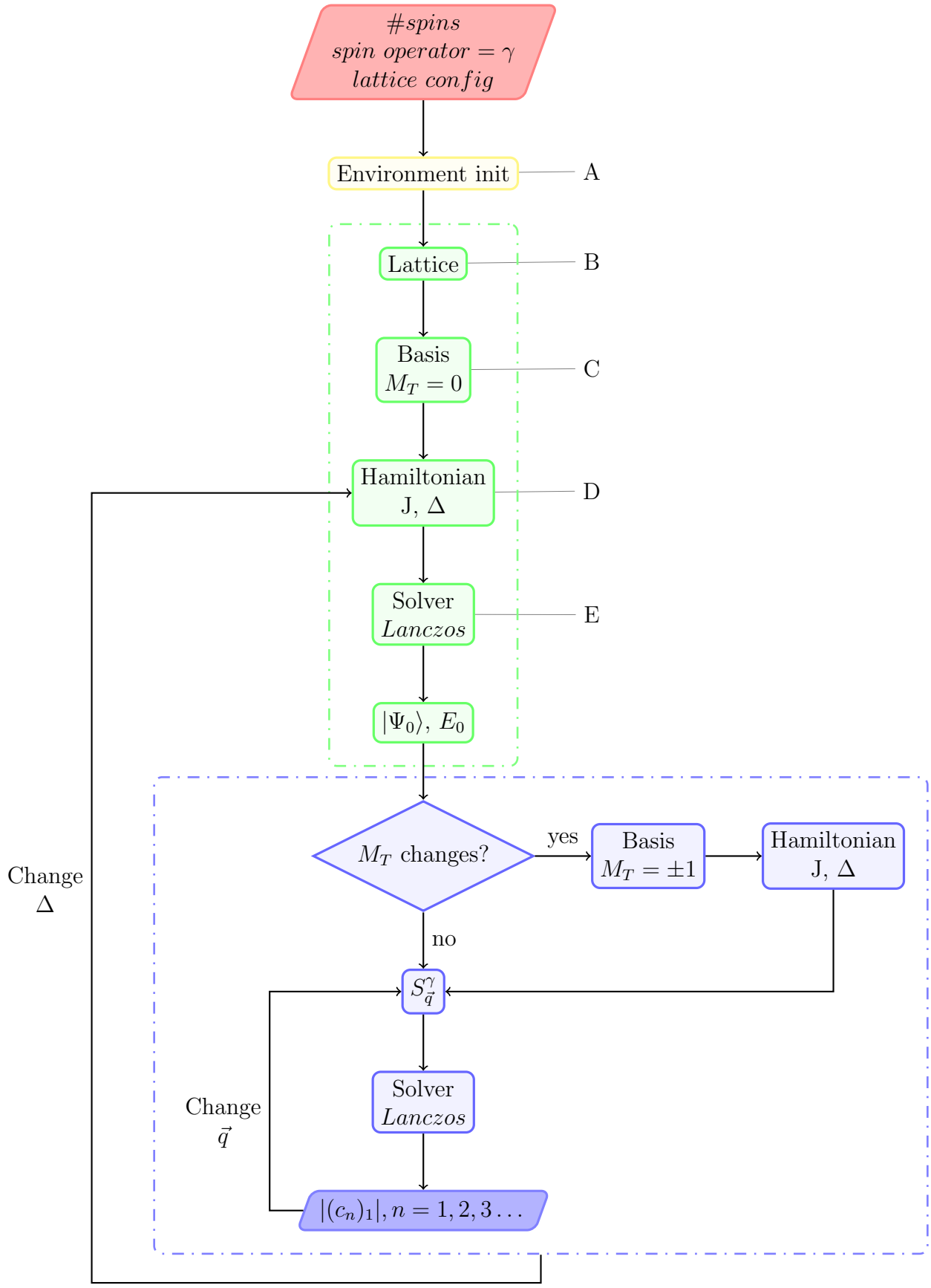


Figure 3.1: Diagram of the final full code.

memory just for the basis elements. A detailed explanation of the construction of the basis elements is provided later in Section 3.2. It is possible to do an exponential fitting and see that the growth of the number of elements in the basis goes as $0.22 * 2^{0.98*\texttt{nspins}}$.

Hamiltonian Matrix

(D) The second step of the Exact Diagonalization procedure is to build the Hamiltonian of the system. This is one of the most time consuming parts of the code up to now. The `Hamiltonian` class requires a `Lattice` and a `Basis` objects to be constructed, and its main purpose is to build the matrix that represents the Hamiltonian of Equation 2.4. The matrix is stored in a `Mat` PETSc's object, specifically it is a `MATMPIAIJ` kind of matrix, which means a parallel sparse matrix. As the matrix is sparse, the total number of elements is not $\mathsf{N}_{\mathcal{H}_{\text{red}}} \times \mathsf{N}_{\mathcal{H}_{\text{red}}}$, still how sparse the matrix is depends on the number of spins in the system and the lattice configuration. All this will be clearer when I discuss the matrix creation in Section 3.3.

Solving

(E) After the matrix is assembled everything is ready for its diagonalization. The solver is initialized and the eigenpairs are obtained. This part is handled by the `Solver` class. For the matrix diagonalization, SLEPc handles all the work, providing that the user correctly initializes and set the solver parameters.

Dynamical Lanczos

The part of Figure 3.1 that is enclosed in a blue rectangle is the one relative to the Dynamical Lanczos procedure that is not coded yet. However, the work completed up to now, in this thesis, was done having in mind that it needs to be coupled with the next part and both have to work together; for example, some of the building blocks described before are also part of this section, like the `Basis`, `Hamiltonian` and `Solver` classes.

3.2 Creation of Basis

How can the system basis be represented? In Quantum Mechanics, once the quantization direction is established, in our case z-direction, the state of a 1/2 spin can be written as a combination of two basis states, $|\uparrow\rangle$ and $|\downarrow\rangle$ or, what is the same, $|+\frac{1}{2}\rangle$ and $|-\frac{1}{2}\rangle$. This 1/2-spin basis states satisfy the following, when applied to spin operators:

$$\begin{aligned} S^z|\uparrow\rangle &= 1/2\ \hbar\ |\uparrow\rangle, & S^z|\downarrow\rangle &= -1/2\ \hbar\ |\downarrow\rangle, \\ S^+|\uparrow\rangle &= 0, & S^+|\downarrow\rangle &= \hbar\ |\uparrow\rangle, \\ S^-|\uparrow\rangle &= \hbar\ |\downarrow\rangle, & S^-|\downarrow\rangle &= 0. \end{aligned} \tag{3.2}$$

In this work $\hbar$ is taken to be 1. In order to understand how the matrix gets constructed, it is important to remark that these are elements of a basis, therefore they satisfy $\langle a|b\rangle = 0$ and $\langle a|a\rangle = 1$.

When more than one spin is involved, it is possible to represent their state as a direct product of the state of each spin. This means that if the individual state of each of the N spins is represented as $|0\rangle$, $|1\rangle$, ..., $|N-1\rangle$, the system state is given by

$$|\Psi\rangle = |N-1\rangle \otimes \cdots \otimes |1\rangle \otimes |0\rangle = |(N-1) \ \ldots \ 1 \ 0\rangle^2. \tag{3.3}$$

In order to be able to use this states to compute anything, the "up" state, $|\uparrow\rangle$, is represented by as a 1 and the "down" state $|\downarrow\rangle$ as a 0. Let us make an example to render things clearer. Think of a system of 6 spins in which half of them are pointing up, meaning they are in the state $|\uparrow\rangle$, and the other half are pointing down, starting with an "up" spin. Let's assume also that the spins in the lattice are numbered and contiguous spins point in opposite directions. The lattice in itself is not something that plays a role in the basis definition, it is only important to know that spins are numbered, in some way, from 0 to $\texttt{nspins} - 1$. After all the setting, the system state is given by $|\,010101\rangle$. By using 1's and 0's for the states they can be treated as bits, and the state of the whole system can be seen as the bit representation of a positive integer number. In this example, the state of the system is represented by the number 21.

In order to create the basis, one has to know how many spins "up" and "down" there are, and for that two quantities are needed: total number of spins in the system and the total magnetization M_T. Then it is possible to obtain the number of spins "up", $N_\uparrow$, as

$$N_\uparrow = \frac{2M_T + \texttt{nspins}}{2}; \tag{3.4}$$

the number of spins "down" is just $N_\downarrow = \texttt{nspins} - N_\uparrow$. Once this is done, the basis elements are the integer numbers whose bit representation are all the possible combinations of $N_\uparrow$ 1's and $N_\downarrow$ 0's. For example, considering 6 spins with $M_T = 1$ the basis has 15 elements (see Table 3.2).

There exists an algorithm, called Gosper's hack [15], that uses bitwise operations and gives the integer numbers in an ascending order, and it is depicted in Algorithm 3.1.

Algorithm 3.1: Gosper's Hack

```
input: int x (one element of the basis)
output: int y (next element of the basis)

unsigned long u = x & -x;
unsigned long v = u + x;
x = v +(((v^x)/u)>>2);
```

[2]The order of the spins in the system state is displayed in descending order, because it is related to how the computer indexes the bits of a number when using its bit representation.

In the code, this algorithm is used in a ($\mathsf{N}_{\mathcal{H}_{\mathsf{red}}}$)-step loop starting with the smallest integer that one can create with $N_\uparrow$ 1's and $N_\downarrow$ 0's. The last integer number, the biggest one one can create with the specified number of spins "up" and "down", is known before looping through the algorithm, this allows us to avoid an `if` statement to check if there is an overflow in `v`. The elements are stored in an `std::vector<PetscInt>`, as already mentioned, in ascending order, therefore to each element corresponds an index ranging from 0 to $\mathsf{N}_{\mathcal{H}_{\mathsf{red}}} - 1$.

index	bits	integer
0	001111	15
1	010111	23
2	011011	27
3	011101	29
4	011110	30
5	100111	39
6	101011	43
7	101101	45
8	101110	46
9	110011	51
10	110101	53
11	110110	54
12	111001	57
13	111010	58
14	111100	60

Table 3.2: Basis elements, bit and integer representation, for a 6 spins system with $M_T = 1$.

3.3 Creation of Hamiltonian

As mentioned before, the construction of the Hamiltonian is one of the most time consuming parts of the code up to now. It consists in two main parts: preallocation and setting values.

Preallocation

One very important detail about the PETSc library when creating `Mat` objects is the *preallocation*. Creation of a `Mat` object has three required steps: first, a call to the function `MatSetSizes(Mat,...)`; second, the elements specification, that includes one or many calls to the function `MatSetValues(Mat,...)`, where one assigns values to the non zero elements of the matrix; and finally, the assembly, a call to the pair of functions `MatAssemblyBegin(Mat,MAT_FINAL_ASSEMBLY)` and `MatAssemblyEnd(Mat,MAT_FINAL_ASSEMBLY)` that perform the necessary communications between MPI processes and leave the `Mat` object in the correct state for

it to be used. When dealing with sparse matrices, not all the memory stated in the creation is used, so in principle, it is not allocated at the beginning. This means that one is using that memory dynamically, which generates an overhead in the calls to the `MatSetValues(Mat, ...)` function. For this matter, preallocation is needed. If one is able to provide the sparsity pattern, the place where the non zero elements of the matrix are, before filling the matrix, the allocation is done at the beginning and memory is no longer treated dynamically. From Table 3.3, it is visible that, while the creation of the Hamiltonian is only less than two times slower when preallocating, the building of the off-diagonal part is 80 and 135 times faster, for the case of 26 and 28 spins, respectively.

# spins	hamilt (s)			offdiag (s)		
	no	yes	ratio	no	yes	ratio
26	1.103	2.274	0.48	138.7	1.709	81.15
28	3.261	8.003	0.40	817.8	6.031	135.6

Table 3.3: Execution time (in seconds) of the creation of the Hamiltonian (`hamilt`) and the building of the off-diagonal part (`offdiag`) with and without preallocation, for 26 and 28 spins, run with 16 nodes.

Hamiltonian elements

The Hamiltonian in Equation 2.4 has the symbol $\sum_{\langle i,j\rangle}$, this means that one has to sum the quantity that follows it over all the pairs of spins i, j that are nearest neighbours, meaning contiguous in numbering. Now it is clear that the lattice becomes an important element as one needs to know who is neighbour of whom. Each of the elements of the matrix corresponds to the operation

$$\mathcal{H}_{(m,n)} = \langle \Psi_m | \mathcal{H} | \Psi_n \rangle = \Psi_m^\dagger * \mathcal{H} * \Psi_n, \tag{3.5}$$

that is equivalent to the multiplication of the matrix by the basis element vector on both sides. The said Hamiltonian has two distinct parts: a diagonal and an off-diagonal part. Without paying attention to the constants, the parts are given by $S_i^z S_j^z$ and $S_i^+ S_j^- + S_i^- S_j^+$, respectively.

Suppose a system of 6 spins with total magnetization $M_T = 1$, the basis elements are shown in Table 3.2. Take element 8, $101110 \equiv 46$, and a `1D chain` lattice. The diagonal part is computed as:

$$\begin{aligned}\mathcal{H}_{diag}|101110\rangle = {} & S_0^z S_1^z|101110\rangle + S_1^z S_2^z|101110\rangle + S_2^z S_3^z|101110\rangle + \\ & + S_3^z S_4^z|101110\rangle + S_4^z S_5^z|101110\rangle + S_5^z S_0^z|101110\rangle.\end{aligned} \tag{3.6}$$

Each term gives the following result:

$$\begin{aligned} &S_0^z S_1^z|101110\rangle = (-1/2) * 1/2\ |101110\rangle, && S_1^z S_2^z|101110\rangle = 1/2 * 1/2\ |101110\rangle, \\ &S_2^z S_3^z|101110\rangle = 1/2 * 1/2\ |101110\rangle, && S_3^z S_4^z|101110\rangle = 1/2 * (-1/2)\ |101110\rangle, \\ &S_4^z S_5^z|101110\rangle = (-1/2) * 1/2\ |101110\rangle, && S_5^z S_0^z|101110\rangle = 1/2 * (-1/2)\ |101110\rangle, \end{aligned}$$

therefore, Equation 3.6 gives for the element $(8,8)$ of the Hamiltonian matrix

$$\mathcal{H}_{diag}|101110\rangle = -\frac{1}{2}\,|101110\rangle = \mathcal{H}_{(8,8)}. \tag{3.7}$$

As the spin operator S_i^z does not change the basis element that is applied to it, the diagonal part only connects basis elements with themselves. To sum up, when two neighbour spins are in the same state, one has to sum 1 to the diagonal value of the basis element it is being considered, and when they are in opposite states one has to subtract one to the diagonal value. At the end, the result has to be multiplied by the constant $-|J|\Delta/4$.

The off-diagonal part has S_i^+ and S_i^- spin operators which change the state of the system when applied to them. Following the same example as before one has

$$\begin{aligned}
\mathcal{H}_{off}|101110\rangle = &\left(S_0^+S_1^-|101110\rangle + S_0^-S_1^+|101110\rangle\right) + \\
&+ \left(S_1^+S_2^-|101110\rangle + S_1^-S_2^+|101110\rangle\right) + \\
&+ \left(S_2^+S_3^-|101110\rangle + S_2^-S_3^+|101110\rangle\right) + \\
&+ \left(S_3^+S_4^-|101110\rangle + S_3^-S_4^+|101110\rangle\right) + \\
&+ \left(S_4^+S_5^-|101110\rangle + S_4^-S_5^+|101110\rangle\right) + \\
&+ \left(S_5^+S_0^-|101110\rangle + S_5^-S_0^+|101110\rangle\right),
\end{aligned} \tag{3.8}$$

$$\begin{aligned}
\mathcal{H}_{off}|101110\rangle = &\left(|101101\rangle + 0\right) + \left(0 \;\; + 0\right) + \left(0 \;\; + 0\right) + \\
&+ \left(0 \;\; + |110110\rangle\right) + \left(|011110\rangle + 0\right) + \left(0 \;\; + |001111\rangle\right).
\end{aligned}$$

The off-diagonal part connects basis element 8 with basis elements $7 \equiv 101101$, $11 \equiv 110110$, $4 \equiv 011110$ and $0 \equiv 001111$. In summary, in order to get the off-diagonal part of a row associated to a basis element, one needs to check if two neighbouring spins are in opposite states, swap them and see which basis element matches the state obtained after the swap.

Putting both parts together one obtains the eighth row of the matrix:

$$\mathcal{H}_{(8,:)} = \left(-\frac{|J|}{2}, 0, 0, 0, -\frac{|J|}{2}, 0, 0, -\frac{|J|}{2}, -\frac{|J|\Delta}{4}, 0, 0, -\frac{|J|}{2}, 0, 0, 0\right). \tag{3.9}$$

The way the previous operations are carried out in the code is through bitwise operations (see Algorithm 3.2). The static array `coupled_elems` has the indices of the basis elements to which the elements `x` is connected to after the swaps. The next challenge is the `search` function that given a basis element `swp` has to return the corresponding index `new_index`.

3.3.1 The search function

The searching procedure poses the first communication issue when the memory is distributed. In this case, each process possesses only a part of the basis elements, therefore searching for an element that does not belong to the process doing the

search requires a communication between this one and the process that owns the said element. What is usually done in in this step is a *ring exchange*. It is a procedure in which every process communicates to all the others its part of the data in an ordered way; for example, all processes send their data to the next-in-rank process, the data is used and sent again to the following process, until all processes have "seen" all the data. This requires N^2 communications of an array of 64-bit integers that can reach considerable dimensions (Table 3.1). Other alternative, that may also include communications, is the use of hash tables, which imposes the use of extra memory to store the data structure and also requires a good hash function to make the search efficient.

Algorithm 3.2: Procedure to obtain the elements of the matrix.

```
input: int x (basis element), Lattice lat (lattice
   object)
output: int diag, int[] coupled_elems

int swp, col = 0, new_index;
for i ∈ [0,nspins)
    for n ∈ lat.neighbours
        bitA = (x >> i) & 0x1;
        bitB = (x >> n) & 0x1;

        if (bitA ^ bitB)
            --diag;
            swp = x ^ (1u << i);
            swp ^= (1u << n);
            coupled_elems[col] = search(swp,new_index);
            ++col;
        else
            ++diag;
```

Fortunately, there exists another alternative that does not require nor communications nor memory. It is a *perfect hashing function* [16], which is not easy to find, as it is commonly known. The function makes use of the ordered basis as well as the bit representation of its elements. The index of a basis element is given by

$$I = \sum_{i=1}^{N_\uparrow} \binom{p_i}{i}, \tag{3.10}$$

where p_i is the position of the i-th "up" spin and $\binom{p_i}{i} = 0$ when $p_i < i$. Using again the same setting as before, 6-spins system with total magnetization 1 ($N_\uparrow = 4$), take the elements 4, 7 and 11 as examples; results of the use of the hashing function are shown in Table 3.4.

integer	bits	p_1	p_2	p_3	p_4	index
30	011110	1	2	3	4	4
45	101101	0	2	3	5	7
54	110110	1	2	4	5	11

Table 3.4: An example of the use of the "perfect hashing function" for a 6-spins system with total magnetization 1.

Even though this cuts off the communications entirely, all this operations come with a cost, smaller but a cost after all. The binomial coefficient requires the factorial of the arguments and a division

$$\binom{p_i}{i} = \frac{(p_i - i + 1) \times (p_i - i + 2) \times \cdots \times p_i}{2 \times 3 \times \cdots \times i}, \tag{3.11}$$

and this operation has to be performed many times. How many? Well, that strongly depends on the lattice: for a `1D chain`, each spin has only two neighbours (however, in the code the neighbour pairs are counted only once, so each spin has one effective neighbour); while in the case of a `2D square` lattice, each spin has two effective neighbours (4 in reality), and that allows many more possible swaps. It also depends on the total magnetization: the proportion between $N_\uparrow$ and $N_\downarrow$ enables more swap possibilities; also, it has an effect on the number of terms in the sum of Equation 3.11, the more (positive) magnetization, the larger $N_\uparrow$, the more the terms in the sum. However, in the most promising case, at least $\mathsf{N}_{\mathcal{H}_{\mathsf{red}}}$ searches need to be done (`1D chain` with $N_\uparrow =$ `nspins` $- 1$ and $N_\downarrow = 1$), when dealing with 34 spins that is at least $2,333,606,220$ searches. Taking $M_T = 0$, the number of spins "up" is the maximum one can have and the number of swaps one has to do for each basis element ranges from 2 (for the "extreme" elements, the ones that have all the ones together) to `nspins` (when the ones and zeros are alternated). For 1D systems like the ones being treated in here, the number of elements according to the number of spins can be seen in Table 3.5. The number of searches is

# spins	diagonal elems	off-diagonal elems	total elems
24	1,850,380	33,860,736	35,711,116
26	10,400,600	140,616,112	151,016,712
28	28,337,976	582,433,600	610,771,576
30	155,117,520	2,406,996,000	2,562,113,520
32	435,443,490	9,927,521,280	10,362,964,770
34	2,333,606,220	40,873,466,520	43,207,072,740
36	6,711,230,900	168,019,647,840	174,730,878,740
38	35,345,263,800	689,710,282,800	725,055,546,600

Table 3.5: Number of matrix elements according to the number of spins. The diagonal elements and the off-diagonal elements are specified separately to see their contribution to the total.

equal to the number of off-diagonal elements, and it increases exponentially like

$exp(0.707254 * \texttt{nspins} + 0.3862)$ with the number of spins.

3.3.2 PETSc interface

The Algorithm shown in 3.2 is performed twice for each basis element, once for the preallocation and then for the assembly. Therefore, it is a function (`get_elems`) that is called from two "wrapper" functions that contain the calls to PETSc functions. The general list of calls goes as shown in Algorithm 3.3, where one has to choose between the preallocation calls or the assembly calls for each case.

Algorithm 3.3: Wrapper functions for preallocation and assembly.

```
PetscInt Istart, Iend;
MatGetOwnershipRange(hamilt, &Istart, &Iend);
for (PetscInt elem = Istart; elem < Iend; ++elem)
    get_elems(basis[elem], lat, diag, coupled_elems);

    // Preallocation:
    prealloc_info(···); // User-defined function

    // Assembly
    MatSetValues(hamilt,···); // PETSc function

// Preallocation
MatMPIAIJSetPreallocation(hamilt,···); // PETSc function

// Assembly
MatAssemblyBegin(hamilt,MAT_FINAL_ASSEMBLY); // PETSc
   function
MatAssemblyEnd(hamilt,MAT_FINAL_ASSEMBLY); // PETSc
   function
```

3.4 Solver

Once the matrix is assembled, the problem is left to SLEPc for it to solve it and get the converged eigenpair, by calling the functions

```
EPS solver;
PetscInt nconv;

EPSCreate(PETSC_COMM_WORLD, &solver);
EPSSetOperators(solver,matrix,NULL);
EPSSetProblemType(solver, EPS_HEP);
```

```
EPSSetWhichEigenpairs(solver, EPS_SMALLEST_REAL);
EPSSetFromOptions(solver);

EPSSolve(solver);
EPSGetConverged(solver,&nconv);
EPSGetEigenpair(solver,...);
EPSDestroy(&solver);
```

This is the normal setup for a SLEPc program. The user is able to tweak the `solver` object during runtime because of the presence of the call to the `EPSSetFromOptions` function.

3.5 Software Development Tools

A consistent and efficient code is not the one that only brings the user to "immediate" results. There are a set of other features that make it reliable, reusable and easily modifiable and extendable. This section takes a look on why it is important to consider these characteristics when developing a code.

Reliability is needed when one plans to make hypothesis or get conclusive results from the software, it is of great concern that the code provides repeatable results at every stage of its development and that it gives correct results for already known situations; this approach also makes the eventual debugging easier, as one makes check points continually. Such thing can be accomplished through unit tests and integration tests, the code developed for this thesis has both of them.

Each individual function has been tested for small known cases using `Google Test`, a test platform. It provides easy-to-use environments and functions to make testing a simple, effortless, task and to allow the developer to concentrate exclusively on the test content rather than the machinery needed to make it work correctly. The basic usage is through "assertions" to verify the tested code's behavior, and with this a great deal of thing can be accomplished already. In addition, the overall code functioning, meaning the value of the ground state energy of the system, has also been tested using integration tests, moreover results have been compared to literature values [17] and with results obtained with Monte Carlo Method [18].

Reusability is a concept that shapes the whole developing environment, it does not only affect the coding. It brings together many aspects like *building*, *installation*, *configuration*, *deployment*, *maintenance* and *upgrading*; this project explores some of them, while others are prospective features to be include.

The most common and widely used building tool is `CMake`, however, in this project another build system tool named `Meson` is used. It is very user-friendly, with clear, reasonable and common-sense syntax, similar to Python's one, which makes it pretty easy to use. Among others, a great characteristic of this build system tool is the simple way of integrating dependencies, like libraries and include files, that is usually a painful and cumbersome task.

When in comes to the code in itself, reusability has to do also with modularity and how small parts sum up to build larger parts. This improves readability and makes it less error-prone, also it is simpler to debug and test. Moreover, it has great impact in the time spent when adding more features, modifying it and upgrading it. This is accomplished in this code in two ways: by writing small functions, that perform a simple, specific task and prevent code duplication; and by the usage of independent classes and namespaces, which organize the code into layers.

In general, software is developed by many people and in different stages. In particular, scientific codes are used and built by several generations of researchers inside the same group or even in different groups, which means that the person that wrote some parts may not be present anymore for consultations. These facts require the code to satisfy all the previous mentioned characteristics, but also it is essential that it is documented. For this matter, `Doxygen` is used. It generates the documentation in either an HTML or a PDF (among other formats) from commented, formated code in the source files.

On top of all this, a version control system, `Git`, is used. This allows all the participants of a project to keep track of the changes made in the files; it also permits to keep different versions of the code, that comes in handy when implementing new features or repairing a bug. It generates a backup of every past state of the code enabling the developer to go back in time and come back without loosing any information. In addition, the code is hosted in the `GitHub` server, that not only offers the same functionalities as `Git`, but also provides very useful collaboration features and allows developers to share their product with the community, key aspects in software development.

Finally, there is another tool worth mentioning: a continuous integration (CI) service, `Travis CI`, that is used in combination with `GitHub`. It enables the developer to check, when pushing to the `GitHub` server, that the changes made in the commit are consistent with and can be integrated without conflict to the already existent code. In cases like this, where libraries with a particular configuration are needed, setting the CI tool is not trivial. As for that, a Docker container, starting from an Ubuntu 18.04 image, was set up. This container is downloaded and executed when starting the CI test.

Chapter 4

Tests and Performance

There is a common phrase that we hear repeatedly during the Master and that one should maybe take as a mantra when developing a code, specially when giving the first steps in software development: DO NOT TRY TO OPTIMIZE THE CODE BEFORE MAKING IT WORK. However, there is a crucial thing missing in that phrase. Once the code has reached a desired state, once the "making it work" part has passed, it is also necessary to PROFILE THE CODE AND TO TEST ITS PERFORMANCE BEFORE TRYING TO OPTIMIZE IT. It is not wise to modify some part of the code with only the hunch or belief that it is there where the lack of performance resides, one has to check which are the major bottlenecks and start from there, otherwise a significant amount of time may be lost in the process of optimization, making the "high performance" concept very contradictory.

With this goal in mind, we have tested code in an HPC facility, we have performed scalability tests and measurements of memory consumption, and we have compared the performance of the solvers chosen [1]. In this chapter, we present the results obtained from the tests.

4.1 System Topography

The code was run on the A2 partition of the Marconi machine at CINECA center. This partition is provided with Intel Xeon Phi 7250 processors (commonly known as Knight Landing - KNL) and Intel Omnipath 100 GB/s network:

- Number of cores per node: 68 (with Simultaneous MultiThreading)
- Frequency: 1.4 GHz
- Memory per node: 16 GB of MCDRAM + 98 GB of DDR4
- Caches: 1 MB L2 shared between two cores, 32 kB L1d and L1i.

[1] In order to do all the performance tests, and to later obtain results regarding the physics problem, a large amount of CPU hours is required. 4 million CPU hours on Marconi A2 were granted to us by CINECA through the *Italian SuperComputing Resource Allocation - ISCRA* after submitting a type C proposal for this project and being accepted.

- Instruction set extension: AVX-512

Something to comment about this particular processor is the memory layout. The Multi-Channel DRAM (MCDRAM) can be configured in three different modes. As there is no shared L3 cache memory in KNL, one can configure MCDRAM as a unusually large L3 cache (*cache* mode), or as part of the DDR memory (*flat* mode), or even in a *hybrid* mode, in which part of it works as plain DDR memory and part as L3 cache. Each setting differs significantly in the metrics of capacity, bandwidth and latency, considering that MCDRAM can achieve a peak bandwidth of approximately $400\,\mathrm{GB\,s^{-1}}$, while DDR only delivers around $90\,\mathrm{GB\,s^{-1}}$ [19].

The default mode, and also the mode set on Marconi, is the cache mode. With this configuration the memory is managed entirely by the hardware and is transparent to the user. This allows its use without the need of any modification of the code or manual allocation of data structures. The flat mode consists in both memories working together as an heterogeneous main memory. The downside of this mode is that it requires either code modifications or using `numactl` to control the data placement. The hybrid mode, as its name claims, consists in splittings the MCDRAM into cache and DDR, but it requires the reboot of the system and modifications of the BIOS.

Results obtained in [19] show that in cache mode the bandwidth depends largely on the data size. There is a peak performance of $260\,\mathrm{GB\,s^{-1}}$ when the size is approximately 8 GB, after this it drops considerably reaching a bandwidth below $50\,\mathrm{GB\,s^{-1}}$ when the size of the data is larger than 40 GB. It was an early goal to try to fit the problem in the 16 GB memory corresponding to MCDRAM, even if it was configured in cache mode. After monitoring the memory consumption, we came to learn that our system's consumption reaches 40 GB per node in some cases, which prevents us from exploiting the features of the on-chip memory present in KNL. Memory analysis will be presented further in this chapter.

4.2 Libraries' configuration

As mentioned before, the libraries used play a very important role in the code performance, as they are the ones managing the parallel distribution of the data and diagonalizing the matrix. Is because of this that its configuration is of great relenvance.

The version of PETSc and SLEPc used are 3.10.2 and 3.10.1, respectively. The settings are done for PETSc's installation, later SLEPc automatically takes the same configuration options (as it is built on top of the first one). They have been compiled using Intel compiler and Intel's MPI version; in addition, Intel's MKL math library was used for the algebraic operations, such as matrix-vector or matrix-matrix multiplications. All these components are the ones corresponding to the 2018 version update 4. Configuration options explicitly used (apart from the default ones) are listed below:

- `--with-precision=double`
- `--with-scalar-type=real`
- `--with-64-bit-indices=1`
- `--with-shared-libraries=1`
- `--with-avx512-kernels=1`
- `--with-memalign=64`
- `--with-debigging=0`
- `--with-mpi=1`
- `--with-mpi-compilers=1`
- `--with-default-arch=0`
- `--with-valgrind=1`
- `--with-batch=1`
- `--known-mpi-shared-libraries=1`
- `--known-64-bit-blas-indices=1`
- `--CC=mpiicc, --CXX=mpiicpc, --FC=F90=F77=mpiifort`
- `--COPTFLAGS=CXXOPTFLAGS=FOPTFLAGS="-O3 -g"`
- `--CFLAGS=CXXFLAGS=FFLAGS="-DMKL_ILP64 -I${MKLROOT}"`
- `--CC_LINKER_FLAGS=CXX_LINKER_FLAGS=FC_LINKER_FLAGS="-L{MKLROOT} -lmkl_intel_ilp64 -lmkl_sequential -lmkl_core -lpthread -ldl"`
- `--with-blaslapack=1`
- `--with-blaslapack-pkg-config=/.../mkl_dynamic_ilp64_seq.pc`

4.3 Performance Analysis

The main performance analysis consists basically on studying the scalability of the application. This means how well or badly the total execution time of the code changes when increasing the workload and/or the resources available. This is an essential step because it allows one to estimate the computational cost and the time needed to execute the code for a given system size. This is a decisive factor, as the outcome of the estimation provides the answer to the question: is it possible to simulate a system with N spins given the resources available?

There are two kind of scaling tests: strong scaling and weak scaling. In both cases one measures the execution time of the whole code, or even of each part separately. In the first test type, the workload (the amount of data) is fixed and the computational resources (number of cores) increase; in the second case, the size of the problem grows proportionally to the computational resources.

From the strong scaling test, it is possible to measure the *speedup* of the application given by the ratio between the time it takes for the program to run in serial, meaning only one process, and the time it takes to run it in parallel with p processes,

$$S = \frac{T_{serial}}{T(p)}. \tag{4.1}$$

In a perfectly parallelizable application, the speedup should be 1 for every value of p, however this is rarely achieved as one needs to take into account other factors. For example, if the code has one or more parts that are not paralellizable at all, the total execution time will be conditioned by this serial part that cannot be reduced no matter how many processes are working at the same time (Amdahl's lay); likewise, when the application requires communication this can have an unfavourable effect on the execution time, as it can create an overhead if it is not done efficiently.

In general, weak scaling is what allows the user of an application to make predictions. When studying complex systems, the more the variables present, the more the computational effort (and memory consumption) to solve it. Usually, one starts by simulating a smaller, more accessible system, however the goal is to be as close to reality as possible. This means that being able to predict the execution time and memory consumption of the application for larger systems is of primary importance. The predictions will permit one to evaluate how many resources are needed to tackle a specific size and if those are accessible.

Considerations regarding the plots in this chapter

- We have chosen to use the Krylov-Schur algorithm for our problem, thus all result presented here, unless indicated otherwise, were obtained with that solver.

- We decided to only use 32 out of the 68 cores available in each node. When using 1/4 of the physical cores of each node the performance is bad compared to the case of using 1/2 or the entire amount of physical cores. In most of the cases the difference in performance between these two last configurations is not significant but the resources used are doubled when using all the physical cores.

- Each result is the outcome of the average of five executions.

In all cases in which choices have been made among a group of possibilities, results backing up the decisions are going to be shown further in this chapter.

Scaling results

In Figure 4.1 it is possible to see the weak scaling of the code. The number of spins goes from 32 to 38, and the number of nodes from 128 to 1024. It can be appreciated from the fitting curve that the scaling is not linear neither quadratic, it goes as

$$0.0207 * x^{1.66}.$$

Being able to have this result is of great importance, as stated before, because now it is possible to predict the total execution time for 38 spins, which should be 2151.02 s.

Looking at the top plot of Figure 4.1, one sees that the creation of the Hamiltonian gets more and more predominant, time-wise, when growing in number of spins and nodes. In order to try to understand this behavior we study the strong scalability for 32, 34 and 36 spins, shown in Figure 4.2.

In this group of plots, one can see that from the significant time consuming parts of the code, the one that scales the best is the building of the off-diagonal part of the Hamiltonian. In a like manner, the solving part presents a good scaling but it decreases the bigger the problem size gets. Contrarily, regarding the creation of the Hamiltonian, the scaling is very poor, and it also becomes more evident for larger numbers of spins. This part is definitely a limiting factor to the scalability in the code. It is evident that for 32 spins running with 512 nodes, it represents more than 3/4 of the total execution time, while for 16 nodes it was 1/3 of the total time. Something to point out with respect to this is that what it is done, computationally speaking, in `hamilt` and `offdiag` is the same. The procedure depicted in Algorithm 3.2 is repeated in both parts, and the only differences between them are shown in Algorithm 3.3.

This unforeseen result led us to use the `perf` tool in juxtaposition with `flame graphs` [20] which allows one to visually check which function is taking more time in the execution. The Figure 4.3a shows the flame graph of the `main` function and a zoom in on the functions `Hamiltonian::Hamiltonian` (constructor) and `Hamiltonian::build_off_diag`, obtained for 26 spins and 4 nodes. From the top left panel of Figure 4.2, it is verified that for small `nspins` and number of nodes the percentage of each part is around 33%, and this is clearly seen in Figure 4.3a.

In order to interpret this picture one has to know that in the y axis is displayed the stack depth, the top box shows the function that was on-CPU (being executed), everything beneath that is ancestry. The width of the coloured boxes shows the total time the corresponding function was on-CPU or part of an ancestry that was on-CPU. All this is based on sample count.

Because the call to the function `boost::math::binomial_coefficient(...)` is done as many times in the `hamilt` part as it is done in the `offdiag` part, our assumption is that the other functions that are called inside them have to be the ones limiting the scaling of the `hamilt` part. For example, the functions `MatMPIAIJSetPreallocation(...)` and `MatCreate(...)` require many calls to MPI functions that are on-CPU for a long time, longer than what the functions

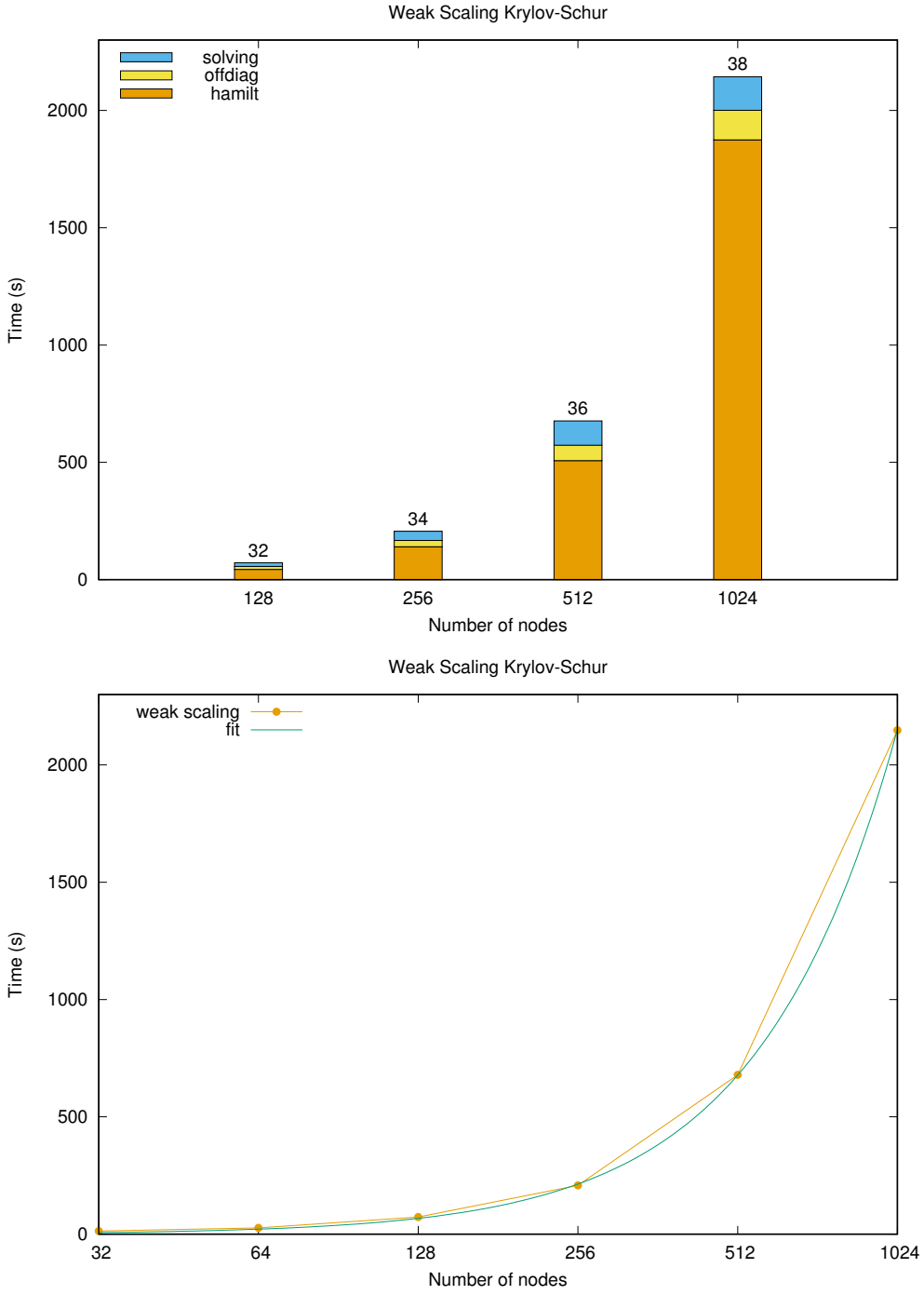


Figure 4.1: Weak scaling using Krylov-Schur solver. Logarithmic base 2 scale on x axis. (top) The contribution of each relevant part of the code to the total execution time. On top of each bar it is shown the number of spins used in each case. (bottom) Total execution time and its fitting: $0.0207 * x^{1.66598}$. For data regarding these plots, see Table A.1.

`MatAssemblyBegin(...)`, `MatAssemblyEnd(...)` and `MatSetValues(...)` are on-CPU.

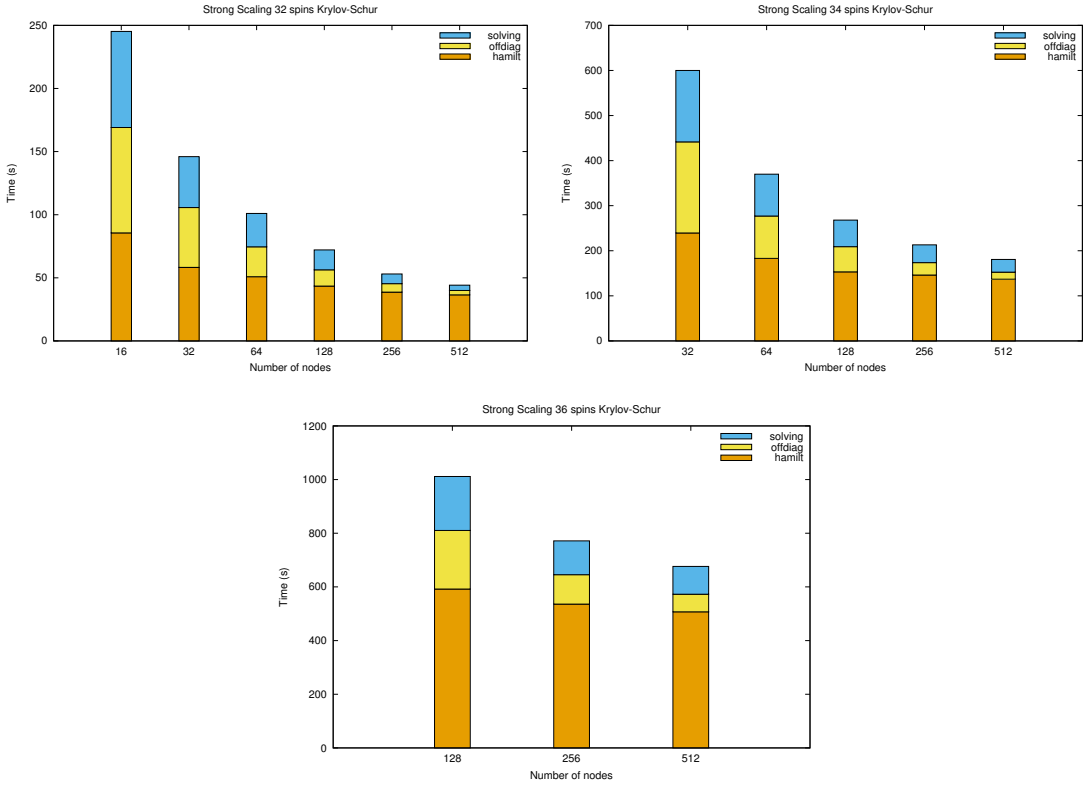
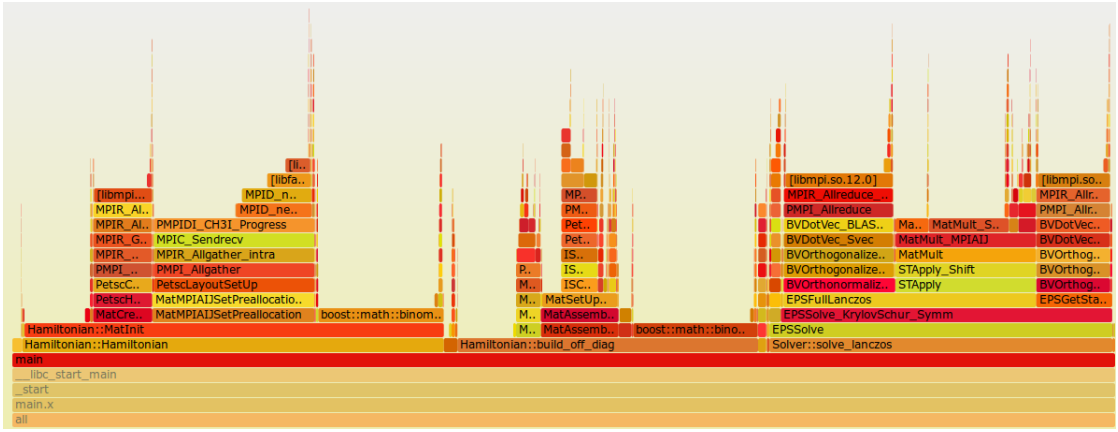


Figure 4.2: Strong scaling for 32, 34 and 36 spins with Krylov-Schur solver. For data regarding these plots, see Tables A.3, A.4 and A.5, respectively.

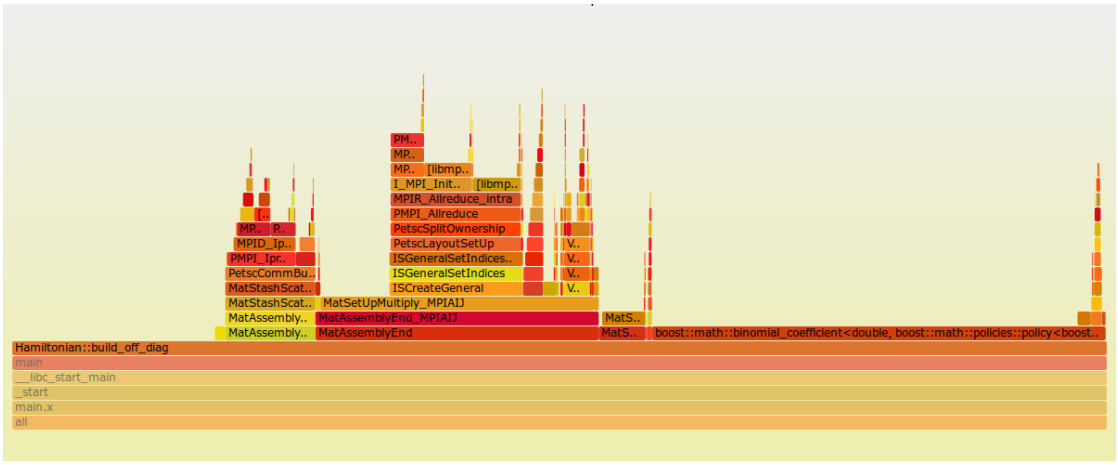
Now thinking about the full code, where the Dynamical Lanczos takes part also, the number of times the `hamilt` part and the `offdiag` part are going to be executed depends on the choice of the S^γ operator: they could be done once (for S^z) or twice (for S^+ or S^-) in one run. The part that will become really relevant at that point is the `solving`, that will be executed as many times as spins are in the system (see Equation 2.3). The speedup of the solver is depicted in Figure 4.4. It was mentioned before, but now it is clearer, that the solver has an almost linear speedup for 32 spins and it decreases for larger system sizes. Because in this part there is a large amount of communications (97% of the total ones approximately, according to PETSc's log report), then the execution time of this part may depend strongly on the number of processes: there must be a good balance between work to do and communications to make. This will be further tested and fine-tuned in order to find the best possible configuration in the next step of the code's development.

The speedup of the less time consuming parts of the code are shown separately in Figure 4.5. In both cases, the reference time of a "serial" run was taken to be the execution time for 128 nodes; in this way, everything was re-scaled according this. In the said Figure, one can see that in both cases, the best speedup, that is almost linear, is for 36 spins. In the case of the creation of the basis, there is a superlinear speedup (it is above the reference line) for 256 nodes for every number of spins displayed here; this behavior also extends to the whole range, in the case



(a) `main` function.

(b) Zoom in on `Hamiltonian::Hamiltonian` function.



(c) Zoom in on `Hamiltonian::build_off_diag` function.

Figure 4.3: Flame graph.

of 32 spins.

The situation in this two parts of the code can be catalogued as an *embarrassingly parallel* one: this means that the problem can be easily parallelized as it requires (almost) none communications. When analyzing the speedup of an

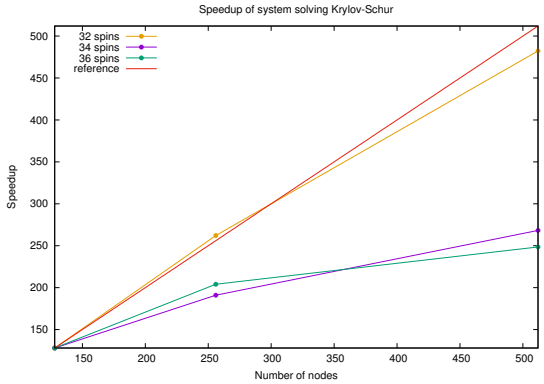


Figure 4.4: Speedup of the `solving` part of the code.

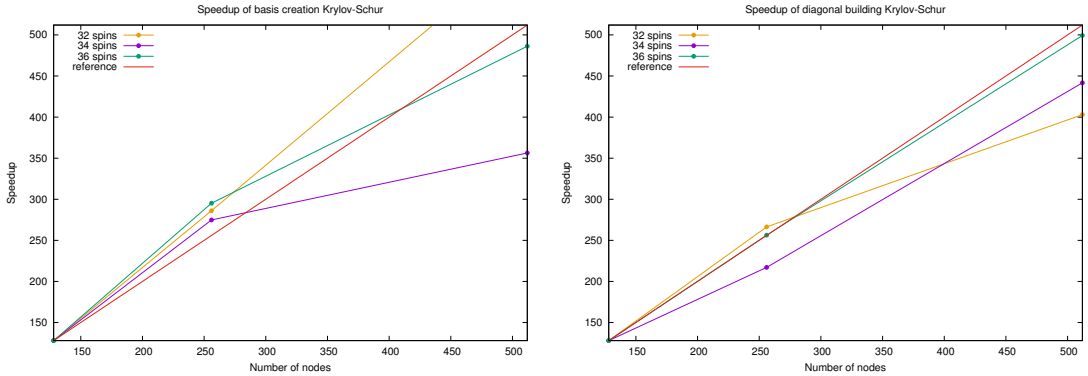


Figure 4.5: Speedup of the basis creation (left) and the building of the diagonal part of the Hamiltonian (right). For data regarding these plots, see Table A.2.

embarrassingly parallel situation, this should be "perfectly linear". During the creation of the basis, no communications at all are performed, however there is a low speedup for 34 spins. Even though this step of the code represents less than one percent of the total time, this is something that gets our attention as its behavior is not as expected. Differently, throughout the building of the diagonal part of the Hamiltonian there is an embarrassingly parallel section, where one sets the values, and then there is a call to `MatAssemblyBegin(...)` and `MatAssemblyEnd(...)` functions that instantiate communications between all the processes, and thit could explain the performance.

Memory consumption results

When a problem is memory bounded, like is in this case, predicting the amount of required memory for a certain system size is as important as predicting the execution time. This was not an easy task during this project. PETSc provides a few functions to monitor the memory like

```
PetscMemoryGetMaximumUsage(...)
PetscMemoryGetCurrentUsage(...)
```

```
PetscMallocGetMaximumUsage(...)
PetscMallocGetCurrentUsage(...)
```

which allow the user to measure the maximum or current resident set size (in the case of the first two) or the maximum or current amount of memory used that was `PetscMalloc(...)`ed during the run (in the case of the second two). It turned out that whatever information those functions were giving back, it did not correspond in any way with the memory consumption inside a computing node, as we came to know by directly logging to one. Nevertheless, we succeeded in measuring the memory consumption and the result for one node is shown in Figure 4.6, for 36 spins running in 512 nodes. In there there are six different stages of the code

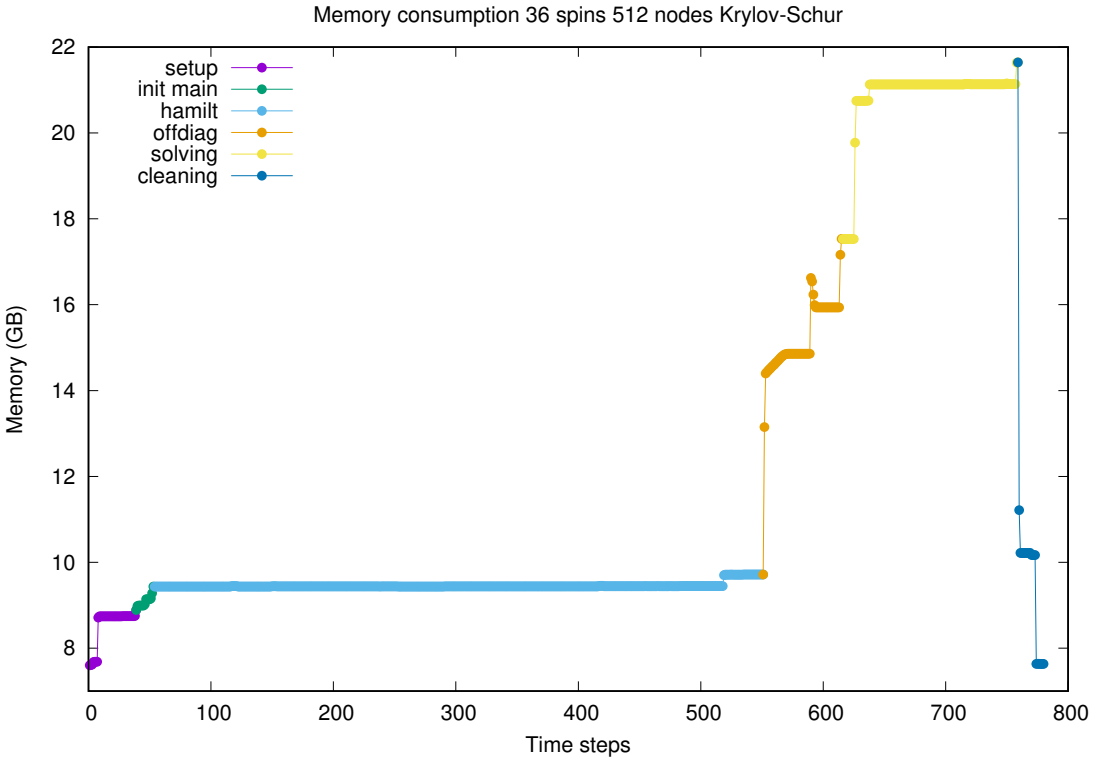


Figure 4.6: Memory profile of one node when running for 36 spins in 512 nodes.

represented by different colors. Some of the stages are too short in time to be able to catch them or differentiate them from the rest, like the basis creation, the lattice creation and the setting of the diagonal values. Among the ones distinguishable, there is at the beginning a `setup` stage, that corresponds to the moment between the `mpirun` command is executed and the initialization of the `main` function; after this, the `init main` stage takes place, in which the SLEPc environment is set up and the creation of the basis and lattice are carried out. Next, the three already well-known time consuming parts, creating the Hamiltonian (`hamilt`), setting the off-diagonal values (`offdiag`) and obtaining the ground state energy of the system (`solving`). An important thing to notice here is the fact that the solver increases by 50% the memory used in the node (taking the initial value to be around 8 GB).

The maximum memory used by a node in this case is 21.63 GB. Assuming that the memory is distributed in a balanced way during the solving part by PETSc, the total maximum memory for 36 spins is 11.07 TB. There are a few minor things to consider about this.

- The memory is not perfectly balanced. By manually checking computing nodes it was seen that there is be a difference of a few gigabytes; it is not possible to specify exactly how many but it was observed at least 5 GB of difference. This could be due to many things; as this is PETSc's domain, it is a bit cumbersome and laborious to sort this out. One thing that could have an influence in the imbalance is the fact that the sparsity pattern of the Hamiltonian is more populated in the middle section of it rather than it its extremes (the more mixed the zeros and ones are, the more the swaps). This results in the processes handling the outer parts of the matrix having less data that the ones handling the middle part.

- The memory is being measured from the moment before the `mpirun` command is executed, and it has been checked that no matter the size of the system or the number of nodes required, there is an initial quantity, around 7 GB of memory, that is always being used before the program is even executed. Therefore, when considering the amount of memory needed to run a job in Marconi A2, that offset of memory needs to be taken into account; while if what one wants to do is estimate the amount of memory used by the code the offset needs to be subtracted from the maximum. In this case, the memory used by the code is ~ 7.5 TB.

Tests on solvers

In the literature, when exact diagonalization problems are treated, the algorithm commonly used is Lanczos, thus we started out using it in our code as well. After an exchange of messages with the SLEPc developers, they strongly suggested us to use their default solver, Krylov-Schur. We found out that Krylov-Schur does indeed perform better than Lanczos, at least for the case of symmetric matrices, therefore, as memory-wise there is no difference between the solvers, we chose to proceeded with Krylov-Schur. It is possible to see in Table 4.1 the execution time of the `solving` part of the algorithm and the iterations performed with both solvers, for different system configurations.

It is clear form the last column of the Table that Krylov-Schur algorithm is faster than Lanczos in delivering the smallest eigenvalue. It is faster not because it performs less iterations before converging, on the contrary, it needs one or two iterations more than Lanczos, but because despite this, it provides the result in less time.

One can also analyze the number of floating point operations per second (*flops* or *flop/s*) for both solvers. In Table 4.2 are exhibited the number of flop provided by PETSc, the execution time for the `solving` part of the code and their ratio. PETSc provides a report on the various stages of the code by enabling the option

# spins	# nodes	time (s)		iters		Ratio
		KS	L	KS	L	times
24	8	0.81	1.15	9	8	1.41
26	16	1.89	2.24	11	9	1.18
28	32	2.93	4.23	11	10	1.44
30	64	5.63	8.79	12	11	1.56
32	128	15.76	33.24	13	11	2.10
34	256	39.14	69.77	13	12	1.78
36	512	103.54	135.57	16	15	1.30

Table 4.1: Execution time for the `solving` part of the code both with Krylov-Schur and with Lanczos.

spins	nodes	solving (s)		TFlop		TFlop/s	
		KS	L	KS	L	KS	L
24	8	0.832	1.155	0.030	0.017	0.0365	0.0147
26	16	1.902	2.246	0.144	0.077	0.0758	0.0347
28	32	2.951	4.240	0.567	0.343	0.1922	0.0809
30	64	5.650	8.828	2.406	1.528	0.4258	0.1730
32	128	15.84	33.30	10.16	6.070	0.6413	0.1822
34	256	39.22	69.85	40.12	26.84	1.0229	0.3842
36	512	103.6	135.6	192.7	133.5	1.8588	0.9846
38	1024	143.1	-	716.3	-	5.005	-

Table 4.2: TeraFlop/s for Krylov-Schur and Lanczos solvers.

`-log_view` during runtime. They count a flop as one real number operation of the type multiplication, division, addition or subtraction. This report is not able to provide information on the functions that are not from PETSc, so it is not possible to retrieve the number of flops performed during the building of the off-diagonal part of the matrix, that includes the binomial coefficient. However, the numbers shown in the last two columns of Table 4.2 give a good idea of the performance of the code.

It is possible to compare this values with the HPCG results of Marconi A2. HPCG is the *High Performance Conjugate Gradients* benchmark [21]. It complements HPL benchmark (High Performance LINPACK) by testing the HPC facilities with a code that "more closely matches a different and broad set of important applications" (see HPCG homepage). Differently from HPL, this benchmark exercises computational and data access patterns that better represent these kind of applications. Among others, it includes sparse matrix-vector multiplication, global dot product and sparse triangular solve. Results for the platform it is used testing are shown in the second column of Table 4.3. Comparing these with the ones obtained for the solving part of the code, it is seen that the performance of the solver is between 10% and 20% of the registered peak performance of the platform.

nodes	TFlop/s	ratio with KS
64	2.490	0.19
128	4.656	0.14
256	9.569	0.11
512	18.254	0.10
1024	32.592	0.15

Table 4.3: HPCG results for Marconi A2.

Chapter 5

Final Remarks

The goal of this project was to simulate systems of frustrated quantum magnets whose Hilbert spaces were larger than $\sim 10^{10}$. This meant, not only to reach the point of what had been done up to now in the field [22], but to go beyond that point. For this, we have built an application in a distributed memory fashion, that relies on two open source scalable parallel libraries, PETSc and SLEPc, both of them capable of handling distributed memory though Message Passing Interface (MPI).

The application consists in modeling the target systems with a Hamiltonian and performing Exact Diagonalization to retrieve the ground state, which is later used as initial vector of the Dynamical Lanczos method. This method consists in finding the eigenvalues of the system with a subspace iteration method for different initial vectors of the subspace, to use them in the calculation of the dynamical structure factor.

We carried out a performance analysis of the application in CINECA Marconi, a Tier-0 supercomputer, particularly in the A2 partition that runs on KNL processors. We have tested the performance of the whole code and of its constituent parts separately. We have observed that the total execution time grows like $0.002 * x^{1.66}$ and that almost every part scales. The construction of the Hamiltonian is the most time consuming of all parts and is the one that does not scale, becoming almost 80% of the total time when the system is 38 spins. This requires our attention in the near future as it is a limiting factor of the speedup of the code. A more thorough analysis of what the PETSc library is doing during this stage may shed some light on this problem. We remark that this limiting factor (which we plan to nevertheless improve upon) becomes quickly irrelevant when moving to Dynamical Lanczos, where allocation will be carried out only once per many diagonalization steps.

Another part of the code requires some of our attention, as its performance is not the expected one. The creation of the basis elements is an embarrassingly parallel task but its speedup is not near the expected linear one for systems with 34 spins.

The solving part of the code is the one that is going the be repeated many times, as many as spins there are in the system, at least. We have tested two different

solvers: the one usually used when dealing with this kind of problems, Lanczos, and the one proposed by SLEPc as the solver that performs the best among the ones present in the library, Krylov-Schur. We have observed a better performance with the latter, and is the one that we will keep using in the future. We have also seen that the solver scales up to 50% of the peak value for systems with more than 32 spins. We have compared the amount of flops given by the solver with the peak performance reported by the machine on the HPGC benchmark, and we found that it corresponds to a 10% to 20%.

We have monitored the memory consumption of the application and we have come to the conclusion that fitting the problem in 16 GB (the MCDRAM equivalent) is not viable. We have observed that there is an initial quantity being used in every node of ~ 7 GB, and that when the solver kicks in, the memory consumption increases up to 50% more in some cases.

Future work

Apart from the things already pointed out in this Chapter, the near future foresees the implementation of the Dynamical Lanczos algorithm and the merging of this with the already available code. This should not present any problems as what has been done up to now was conceived from the beginning to be part of a larger procedure.

Memory is one of the biggest issues in the project and the creation of the matrix is a limiting factor up to this point, therefore a matrix-free implementation may be a future approach. We predict that this will have an impact in the solving part, as each element of the matrix needs to be computed each time it is needed. However, the increase in time of this part may overcome the very expensive, time consuming creation of the Hamiltonian. This has to tested before jumping to conclusions.

Appendix A

Tables with the execution time of the scaling plots of the Krylov-Schur solver of Chapter 4. All the times are in seconds.

spins	nodes	procs	hamilt	offdiag	solving	total	iters
24	8	256	0.793	0.776	0.816	2.476	9
26	16	512	2.274	1.709	1.893	6.017	11
28	32	1024	6.258	3.615	2.937	13.04	11
30	64	2048	14.37	5.909	5.633	26.21	12
32	128	4096	43.46	12.89	15.76	72.72	13
34	256	8192	139.9	27.38	39.14	207.6	13
36	512	16384	507.2	65.63	103.5	678.9	16
38	1024	32768	1874	126.3	142.8	2148	15

Table A.1: Execution time for the weak scaling analysis of the most time consuming parts of the code and the number of iterations perform by the solver (Krylov-Schur). In here 32 processes per node were used.

spins	nodes	procs	basis	lattice	diag
24	8	256	0.0016	4.14e-05	0.0378
26	16	512	0.0027	3.88e-05	0.0807
28	32	1024	0.0064	4.8e-05	0.1408
30	64	2048	0.0090	4.44e-05	0.1854
32	128	4096	0.0151	4.20e-05	0.3315
34	256	8192	0.0280	4.78e-05	0.7700
36	512	16384	0.0461	5.20e-05	1.2699
38	1024	32768	0.0761	4.92e-05	2.8344

Table A.2: Execution time for the weak scaling analysis of the less time consuming parts of the code using Krylov-Schur solver. In here 32 processes per node were used.

nodes	procs	basis	hamilt	diag	offdiag	solving	total
16	512	0.1258	85.59	2.253	83.50	76.09	248.4
32	1024	0.0583	58.27	1.314	47.39	40.33	147.8
64	2048	0.0320	50.89	0.602	23.66	26.40	101.9
128	4096	0.0151	43.46	0.331	12.89	15.76	72.72
256	8192	0.0067	38.67	0.159	6.670	7.698	53.37
512	16384	0.0031	36.48	0.105	3.487	4.184	44.52

Table A.3: Execution time for the strong scaling analysis of the code for 32 spins using Krylov-Schur solver. In here 32 processes per node were used.

nodes	procs	basis	hamilt	diag	offdiag	solving	total
32	1024	0.2200	239.47	4.749	202.14	158.4	606.7
64	2048	0.0886	183.07	2.510	93.989	92.81	373.3
128	4096	0.0603	153.27	1.316	55.901	58.86	270.0
256	8192	0.0280	146.25	0.776	27.438	39.45	214.3
512	16384	0.0216	137.00	0.381	15.631	28.09	181.5

Table A.4: Execution time for the strong scaling analysis of the code for 34 spins using Krylov-Schur solver. In here 32 processes per node were used.

nodes	procs	basis	hamilt	diag	offdiag	solving	total
128	4096	0.1754	592.02	4.953	218.49	200.9	1019
256	8192	0.0760	536.10	2.473	109.56	126.0	775.3
512	16384	0.0461	507.29	1.269	65.638	103.5	678.9

Table A.5: Execution time for the strong scaling analysis of the code for 36 spins using Krylov-Schur solver. In here 32 processes per node were used.

References

[1] C. Lacroix, P. Mendels, and F. Mila, editors. Introduction to Frustrated Magnetism, volume 164 of Springer Series in Solid-State Sciences. Springer-Verlag Berlin Heidelberg, 2nd edition, 2011.

[2] F. Becca and S. Sorella. Quantum monte carlo approaches for correlated systems. Nov 2017.

[3] J. Gubernatis, N. Kawashima, and P. Werner. Quantum Monte Carlo Methods. Cambridge, UK: Cambridge University Press, June 2016.

[4] U. Schollwöck. The density-matrix renormalization group. Reviews of Modern Physics, 77(1):259–315, 2005.

[5] A. Banerjee, J. Yan, Johannes Knolle, Craig A. Bridges, Matthew B. Stone, Mark D. Lumsden, David G. Mandrus, David A., Roderich Moessner, and Stephen E. Nagler. Neutron scattering in the proximate quantum spin liquid α-rucl3. Science, 356(6342):1055–1059, 2017.

[6] B. Lake. Multispinon continua at zero and finite temperature in a near-ideal heisenberg chain. Physical Review Letters, 111(13), 2013.

[7] Yousef Saad. Numerical Methods for Large Eigenvalue Problems, volume 66 of Classics in Applied Mathematics. SIAM (Society for Industrial and Applied Mathematics), 2011.

[8] V. Hernández, J. E. Román, A. Tomás, and V. Vidal. Krylov-schur methods in slepc. Technical Report STR-7, Universitat Politècnica de València, 2007. Available at SLEPc.

[9] V. Hernández, J. E. Román, A. Tomás, and V. Vidal. Lanczos methods in slepc. Technical Report STR-5, Universitat Politècnica de València, 2006. Available at SLEPc.

[10] G. W. Stewart. A krylov–schur algorithm for large eigenproblems. SIAM Journal on Matrix Analysis and Applications, 23(3):601–614, 2002.

[11] James Vance. Large-scale implementation of the density matrix renormalization group algorithm. Master's thesis, MHPC, 2017.

[12] Marlon E. Brenes-Navarro. Parallel implementation of the krylov subspace techniques for unitary time evolution of disordered quantum strongly correlated systems. Master's thesis, MHPC, 2016.

[13] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, Modern Software Tools in Scientific Computing, pages 163–202. Birkhäuser Press, 1997.

[14] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. ACM Trans. Math. Software, 31(3):351–362, 2005.

[15] D. E. Knuth. The Art of Computer Programming, volume 4. Addison-Wesley, 2009.

[16] S. Liang. A perfect hashing function for exact diagonalization of many-body systems of identical particles. Comput. Phys. Commun., 92:11–15, 1995.

[17] Anders W. Sandvik. Finite-size scaling of the ground-state parameters of the two-dimensional heisenberg model. Phys. Rev. B, 56:11678–11690, Nov 1997.

[18] Tiago Mendes Santos. personal communication.

[19] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis. Exploring the performance benefit of hybrid memory system on hpc environments. In 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 683–692, May 2017.

[20] Brendan Gregg. FlameGraph. Accesible through GitHub.

[21] J. Dongarra, M. A. Heroux, and P. Luszczek. Hpcg benchmark: a new metric for ranking high performance computing systems. Technical Report EECS-15-736, Electrical Engineering and Computer Sciente Department, Knoxville, Tennessee, UT, 2015.

[22] Alexander Wietek and Andreas M. Läuchli. Sublattice coding algorithm and distributed memory parallelization for large-scale exact diagonalizations of quantum many-body systems. Phys. Rev. E, 98:033309, Sep 2018.