Master in High Performance Computing

Parallel implementation of the Krylov subspace techniques for unitary time evolution of disordered quantum strongly correlated systems

Supervisor:
Antonello Scardicchio,

Co-supervisors:
Ivan Girotto,
Vipin Varma

Candidate:
Marlon E. Brenes Navarro

2nd edition
2015–2016

Abstract

The study of dynamics of quantum systems proposes both a highly interesting framework to current research in physics and a demanding numerical and computational task. Disordered systems and those that mimic the dynamical properties inherent to disorder, constitute a stepping stone to reach further understanding of quantum electronic and energy transport along with other properties that can be used to shed light to diverse topics in both theoretical and applied physics.
We have developed an application and implemented parallel algorithms using the *Message Passing Interface* in order to provide a computational framework suitable for massively parallel supercomputers to study the dynamics of such physical systems. We used high-performing libraries such as PETSc/SLEPc combined with High Performance Computing approaches in order to study systems whose subspace dimension is constituted by over 9 billion independent quantum states. Moreover, we provide descriptions on the parallel approach used for the two most important stages of the computation: constructing a matrix representation for a generic Hamiltonian operator and the time evolution of the system by means of the Krylov subspace methods. We have enabled the application and successfully performed simulations using three different supercomputers (on both SISSA and CINECA computational frameworks) and provide results to evaluate the overall performance of the application, as well as physical results from the dynamics of a quasi-disordered system under the Aubry-André model.

Acknowledgements

First, I would like to thank my professor and advisor Ivan Girotto, whose insight and skill know no bounds. His support has been invaluable to me during the entire program. Likewise, I would thank my advisors Vipin Varma and Antonello Scardicchio, both members of the CMSP at ICTP; for their support, knowledge and patience. I'm grateful to them for making my stay at the ICTP richer and even more interesting since they introduced me to the topic.
I would also like to thank my friends Peter Labus and Juan Carmona, for discussions and their friendship.
Last, but certainly not least, I thank Marcela Apuy; for the many times she's been there to cheer me up.

The work presented in this thesis and the permanence in Trieste were supported by the Abdus Salam International Centre for Theoretical Physics (ICTP).

Marlon Brenes

December 2016
Master in High Performance Computing

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**

2 Background **3**
2.1 The quantum N body problem 3
2.2 Hardcore bosons 3
2.3 Basis representation 4
2.4 Krylov subspace methods 5
2.5 Towards a massively parallel application 5

3 Workflow and description of the implementation **7**
3.1 Workflow 7
3.2 Configuration of computational dependencies 9
3.2.1 Boost 9
3.2.2 PETSc 9
3.2.3 SLEPc 10
3.3 Serial implementation 11
3.3.1 Construction of the Hilbert space basis representation 11
3.3.2 Construction of the Hamiltonian matrix 12
3.3.3 Time evolution: Krylov subspace methods approach 14
3.4 Parallel version 16
3.4.1 Design 16
3.4.2 Replicated basis version 18
3.4.3 Node communicator version 23
3.4.4 Ring exchange version 28
3.4.5 Combined Ring-Node communicator version 30

4 Results, performance and discussion **36**
4.1 Results 36
4.1.1 Survival probability 36
4.1.2 Disordered systems 36

4.1.3 A perspective on the absolute error 41
4.2 Performance 43
4.2.1 Systems 43
4.2.2 Compilers and libraries 43
4.2.3 Performance results 43
4.2.4 Strong scaling 49

5 Future Work **54**

6 Conclusion **55**

References **57**

A Interacting Aubry-André model results **59**

List of Figures

2.1 Some of the components that take part of the PETSc and SLEPc libraries. Taken from [1] 6

3.1 Workflow description of the problem 8
3.2 Brief summary of the parallel design 17
3.3 Visual representation of the node communicator version 24
3.4 Visual representation of the ring exchange version 29
3.5 Visual representation of the combined Ring-Node communicator version 31

4.1 Survival probability for different system sizes on a system with no disorder 37
4.2 Temporal autocorrelation for the Aubry-André model non-interacting case with 1 particle ($L = 55$) using periodic boundary conditions for different values of λ 38
4.3 Temporal decay exponent γ for different number of particles (N) and values of λ for the interacting Aubry-André model with periodic boundary conditions and $L = 55$ 39
4.4 Temporal decay exponent γ for different densities ($\rho \equiv N/L$) and values of λ for the interacting Aubry-André model with periodic boundary conditions and $L = 55$ 40
4.5 Representation of the absolute error for the time evolution of an unperturbed clean system as a function of the subspace dimension of the quantum system 41
4.6 Running time of the application for the construction of Hamiltonian, time evolution and overall time using a system with $L = 28$ at half-filling (subspace dimension of 40 116 600) up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (replicated basis version) 45
4.7 Running time of the application for the construction of Hamiltonian, time evolution and overall time using a system with $L = 28$ at half-filling (subspace dimension of 40 116 600) up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (`Node comm` version) 47

4.8 Running time of the application for the construction of Hamiltonian and time evolution using a different problem sizes at half-filling up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (`Ring exchange` version) 48
4.9 Running time of the application for the construction of Hamiltonian and time evolution using a different problem sizes at half-filling up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (`Combined Ring-Node comm` version) 50
4.10 Speedup of the application for the construction of Hamiltonian and time evolution using a system with $L = 28$ at half-filling (subspace dimension of 40 116 600) up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (replicated basis version) 51
4.11 Speedup of the application for the construction of Hamiltonian and time evolution using a system with $L = 28$ at half-filling (subspace dimension of 40 116 600) up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (`Node comm` version) 52
4.12 Speedup of the construction of the Hamiltonian step using a system with $L = 30$ at half-filling (subspace dimension of 155 117 520) for a different amount of computing nodes, referenced to two computing nodes (`Node comm` version) 53

A.1 Temporal autocorrelation for the Aubry-André model interacting case with 2 particles ($L = 55$, $\mathcal{D} = 1485$) using periodic boundary conditions for different values of λ (200 realisations) 60
A.2 Temporal autocorrelation for the Aubry-André model interacting case with 3 particles ($L = 55$, $\mathcal{D} = 26235$) using periodic boundary conditions for different values of λ (200 realisations) 60
A.3 Temporal autocorrelation for the Aubry-André model interacting case with 4 particles ($L = 55$, $\mathcal{D} = 341055$) using periodic boundary conditions for different values of λ (100 realisations) 61
A.4 Temporal autocorrelation for the Aubry-André model interacting case with 5 particles ($L = 55$, $\mathcal{D} = 3478761$) using periodic boundary conditions for different values of λ (25 realisations) 61
A.5 Temporal autocorrelation for the Aubry-André model interacting case with 6 particles ($L = 55$, $\mathcal{D} = 28989675$) using periodic boundary conditions for different values of λ (5 realisations) 62

List of Algorithms

3.1 Computing basis size 11
3.2 Next permutation of bits 12
3.3 Binary to integer 12
3.4 Construction of the Hamiltonian matrix 13
3.5 Binary lookup 14
3.6 Parallel distribution 19
3.7 Parallel Construction of the Hamiltonian matrix 21
3.8 Parallel allocation details 22
3.9 Time evolution 23
3.10 Construction of the basis in the Node communicator version 25
3.11 Node communicator parallel allocation details of the Hamiltonian matrix 26
3.12 Node communication pattern 27
3.13 Ring communication pattern 30
3.14 Establishing a communicator group: `zero_node_comm_` 32
3.15 Parallel distribution 33
3.16 Ring-Node communication pattern 35

Chapter 1

Introduction

The study of quantum many-body systems out of equilibrium has received a renovated interest. In early stages, it was established that equilibrium states can be effectively described with proper quantum statistical mechanics models. However, the mechanisms of how these states can be reached by local dynamics that follow microscopic laws is less clear.

A sort of dissonance between a microscopic description and one that uses the classical ensembles from statistical mechanics has raised many questions, for example, as in how the dynamics of quantum phase transitions may be described. Another example is the question of how thermodynamics emerges from microscopic quantum mechanics, which has been of interest since the foundation of quantum theory. For the latter, a setting for non-equilibrium dynamics that has been used in recent studies is related to global *quenches*. In a global quench, the system is in an initial state and then an instantaneous modification of the system's parameters is done. Afterwards, the unitary time evolution of the many-body system under some local Hamiltonian is done to study the behavior of specific observables of interest. This constitutes a method that has been used recently to study *thermalization* or *quantum transport*, for instance. [3]

Unitary time evolution of quantum dynamical systems is, both in intellectual and computational terms, a very demanding task. Given the fact that studying quantum many-body systems out of equilibrium allow us to probe questions in the foundation of statistical mechanics and condensed matter theory, provide quantum simulators related to quantum computing as well as other quantum technologies; a platform to perform numerical operations and simulations involving these systems is indeed very important to current research.

In this sense, a High Performance Computing approach can provide a solid framework to perform these simulations in an efficient manner.

The present work intends to establish an instance of an HPC approach used to tackle

this particular problem. Several different algorithms were developed and optimized in order to implement an application suitable to be enabled in massively parallel supercomputers. Some of these algorithms were developed and optimized during the duration of the project in order to solve specific problems related to the computation of unitary time evolution of quantum many-body systems and further parallelized using the *Message Passing Interface*, however, we also make use of well established and efficient libraries to develop a well-performing and portable application, such as PETSc and SLEPc. [1] [11]

A common practice to evaluate the dynamics of quantum many-body systems is to perform full diagonalisation to obtain eigenvalues and eigenvectors, however, for relatively large systems this practice is computationally problematic when it comes to actual computing times. In order to avoid this, we have employed the method of Krylov subspace techniques. We demonstrate that by using the techniques of Krylov subspaces, the unitary time evolution can be performed successfully up to a given numerical tolerance.

A drawback of the mentioned methodology is given by the amount of memory that is required to perform the calculations in a timely manner. Specifically, the basis of the Hilbert space and the matrix representation of the Hamiltonian operator need to be stored in memory for the used algorithm. Up to a certain degree, established by the amount of memory available in a given computational environment, we have overcome this typical barrier by undertaking a paradigm of full distribution by means of a *Message Passing Interface* model and therefore, the study of larger quantum many-body systems can be performed using typical supercomputing resources. The developed methodology constitutes an instance for which a HPC approach proves most useful towards finding solutions to computational barriers in scientific computing.

We provide a brief background on the problem in Chapter 2, while Chapter 3 is entirely devoted to the description of the computational methodology developed and enabled to tackle specific problems related to the problem from the computation point of view. We perform diverse benchmarks on the developed application and provide physical results that can be obtained with the implemented methodology in Chapter 4.

Chapter 2

Background

2.1 The quantum N body problem

In order to solve the quantum N body problem in the framework of a quantum lattice system, sparse matrix algorithms are usually applied in order to solve for the corresponding eigenstates of a given system. Before proceeding with these sparse matrix algorithms a translation of the considered many-particle Hamiltonian needs to be done, in the language of the second quantization, into a sparse Hermitian matrix. This is usually the intellectually and technically challenging part of the project, in particular if we want to take into account symmetries of the problem. [4]

Typical lattice models in condensed matter involve electrons, spins and even spinless particles; and to understand the properties of different materials and quantum mechanical systems, the Hamiltonian operator of the full quantum problem is simplified using generic models, such as the tight-binding model, the Hubbard model and the t-J model. In the present work we have committed our interest in the Heisenberg model for clean systems and the Aubry-André model for quasi-disordered systems.

All these models exploit different symmetries of the system to translate a problem that scales as $O(exp(N))$ into a polynomial scaling, such as $O(N^4)$. [14]

2.2 Hardcore bosons

The Heisenberg/Hardcore bosons model, as well as other models such as the Aubry-André model, allow us to understand the dynamic of strongly correlated systems. We're particularly interested in the time evolution of such systems to understand such dynamics under certain prepared conditions.

As a particular approach to the problem we have focused our attention on the method applied to the Hamiltonian of the so-called *hardcore bosons* arranged in a 1-D lattice with periodic boundary conditions. A hardcore bosons model is similar

to that of the spinless fermions, but without the antisymmetric exchange property. A Hamiltonian operator can be written for such a system [14]:

$$H = -t \sum_{i=1}^{L-1} (c_i^\dagger c_{i+1} + H.c) + V \sum_{i=1}^{L-1} n_i n_{i+1} \tag{2.1}$$

where in such a system the operation $H|\psi\rangle$ returns a linear combination of other eigenstates in the Hilbert space basis, and so a Hamiltonian matrix can be constructed for the specific operator at hand. The result is a (usually very large) sparse Hermitian matrix. In this framework, an initial eigenstate can be prepared to study the behavior as a function of time of the system by means of the Schrödinger equation:

$$i\hbar \frac{\partial}{\partial t} \Psi(\mathbf{r}, t) = H\Psi(\mathbf{r}, t) \tag{2.2}$$

With the corresponding solutions given by:

$$|\Psi_t\rangle = e^{-iHt/\hbar} |\Psi_0\rangle \tag{2.3}$$

2.3 Basis representation

In order to create a matrix representation of the Hamiltonian operator, a proper representation of the basis vectors of the Hilbert space of dimension $\mathcal{D}$ needs to be devised in order to perform operations on the computer. In particular, for the case of the hardcore bosons described before, the dimension of this space is given by $L!/N!(L-N)!$, where L is the linear dimension of the lattice and N is the number of particles.

An approach that can be used consists in assigning an integer value to each of the basis states of the space. In this representation, each of the states correspond to a value in a memory buffer that can be transversed by lookup algorithms. In that sense, the following is an example for the case of $L = 4$ and $N = 2$:

$$\begin{aligned} |0011\rangle &\rightarrow 3 \\ |0101\rangle &\rightarrow 5 \\ |0110\rangle &\rightarrow 6 \\ |1001\rangle &\rightarrow 9 \\ |1010\rangle &\rightarrow 10 \\ |1100\rangle &\rightarrow 12 \end{aligned} \tag{2.4}$$

This is a very powerful mechanism to represent the basis, given that integer values are easier to work with than binary objects. In particular, if the basis is stored as a contiguous memory buffer effective lookup algorithms can be used to search for a specific element; even more so if the elements are sorted.[1]

2.4 Krylov subspace methods

Applying the methodology described, the problem translates to evaluate the exponential of a large sparse matrix for the system. We may employ the technique of Krylov subspaces in order to **avoid** full diagonalisation. The basic idea corresponds to approximate the solution to equation (2.3) using power series. The optimal polynomial approximation to $|\Psi(t)\rangle$ from within the Krylov subspace,

$$\mathcal{K}_m = span\{|\Psi_0\rangle, H|\Psi_0\rangle, H^2|\Psi_0\rangle \ldots, H^{m-1}|\Psi_0\rangle \tag{2.5}$$

is obtained by an Arnoldi decomposition of the matrix $A_m = V_m^T H V_m$ where m is dimension of the subspace, which is much smaller than the dimension of the Hilbert space.
The solution is then given by:

$$|\Psi(t)\rangle \approx V_m exp(-itA_m)|e_1\rangle \tag{2.6}$$

where $|e_1\rangle$ is the first unit vector of the Krylov subspace. The much smaller matrix exponential is then evaluated using irreducible Padè approximations. The algorithm to evaluate the numerical method has been extensively described by R.B. Sidje. [9]

2.5 Towards a massively parallel application

Given the nature of the problem, a research stage is required in order to evaluate the possible high performance libraries that can be used to perform linear algebra operations. In our particular case, support for sparse matrix representations and operations on these objects is a requirement.
Several different libraries with support for sparse and linear algebra operations were considered in order to solve this particular problem, however, with massively parallel computations in mind, we've devoted our implementation in favor of **PETSc**[1] and **SLEPc**[11]. More about both libraries can be found in the references.
These libraries, provide all the required computational background necessary to carry out the unitary time evolution of the system, a few features that are of particular interest to us are listed as follows:

---

[1] For instance, a binary lookup could be used to search for an element of an array of size N with complexity $log(N)$, compared to the complexity of N given by an element-by-element lookup

PETSc

Nonlinear Systems: Line Search | Trust Region | ...

Time Steppers: Euler | Backward Euler | RK | BDF | ...

Krylov Subspace Methods: GMRES | CG | CGS | Bi-CGStab | TFQMR | Richardson | Chebychev | ...

Preconditioners: Additive Schwarz | Block Jacobi | Jacobi | ILU | ICC | LU | ...

Matrices: Compressed Sparse Row | Block CSR | Symmetric Block CSR | Dense | CUSPARSE | ...

Vectors: Standard | CUDA | ViennaCL

Index Sets: General | Block | Stride

SLEPc

Nonlinear Eigensolver: SLP | RII | N-Arnoldi | Interp. | CISS | NLEIGS

M. Function: Krylov | Expokit

Polynomial Eigensolver: TOAR | Q-Arnoldi | Linear-ization | JD

SVD Solver: Cross Product | Cyclic Matrix | Thick R. Lanczos

Linear Eigensolver: Krylov-Schur | Subspace | GD | JD | LOBPCG | CISS | ...

Spectral Transformation: Shift | Shift-invert | Cayley | Precond.

BV: ...

DS: ...

RG: ...

FN: ...

Figure 2.1: Some of the components that take part of the PETSc and SLEPc libraries. Taken from [1]

- Large list of functionality involving of both sparse and dense operations
- Extensive documentation
- Proven to provide good performance in high-end applications
- Flexibility in the use of datatypes, including complex datatypes
- Profiling component
- Parallelism using *Message Passing Interface* paradigm

PETSc and it's extension SLEPc are *huge* libraries with basically all the functionally required to carry out sparse matrix operations and has been proven to perform very well in many different computational systems and architectures. Figure 2.1 shows a small section of the most important components of the libraries.

Chapter 3

Workflow and description of the implementation

3.1 Workflow

In light of what has been described in the previous chapter, a workflow to obtain solutions to the problem at hand is presented in Figure 3.1.

We start with a an abstract 1D lattice quantum system described by the size of the grid, the number of particles present in the system and a given Hamiltonian operator, such as the one presented in Eq (2.1). The size of the subspace[1] is given by $L!/N!(L-N)!$, with a particular boundary conditions this describes the quantum system.[2] The next step in the workflow is to create a representation of the Hilbert space, as described in the first chapter, this can be done using integer values in the computer. The method we use for this section is a lexicographic computation of next bit permutations, this provides a fast way of computing all the possible combinations in a sorted manner, therefore avoiding the requirement of sorting algorithms for later lookup routines.

The construction of matrix representation of the Hamiltonian operator rests on this basis. This requires to apply the Hamiltonian operator to each of the states in the basis to get the correlation among each of the states. As described in the previous chapter this translates into a sparse, Hermitian matrix that is used for the time evolution of a particular system. A sparse storage format is required in this stage, being that the expected sizes of the system subspace are very large.
The preparation of the initial eigenstates needs to be done in consistently with the format of the Hamiltonian matrix. When it comes to the study of disordered systems, many layers of disorder can be introduced to the system in the form of randomness. One way to introduce disorder to the system is to introduce random-

[1] Composed by all the accessible states of the system

[2] It's easy to see that *half-filling*, i.e, $N = L/2$ gives the larger subspace for the present system

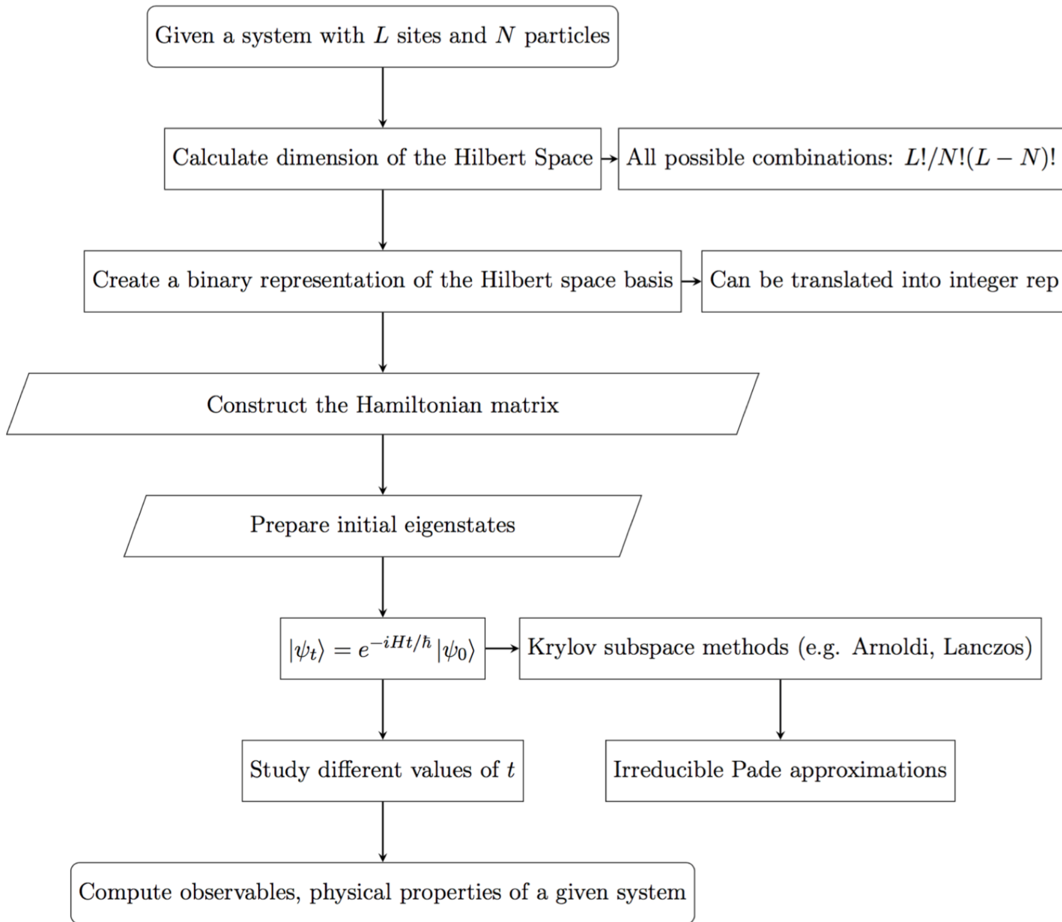

Figure 3.1: Workflow description of the problem

ness to the parameters of the Hamiltonian for example, or to use a random initial eigenstate for each simulation.
When it comes to the unitary time evolution of the system, there's no requirement to construct the Hamiltonian matrix for each timestep. One can just adjust the time parameter and evolve the system using the same Hamiltonian representation. This is done by means of the Krylov subspace methods in order to obtain an evolved state, once this is computed many different properties of the system can be computed in the form of quantum observables, which characterizes the behavior of the system as a function of time.

The following sections provide a very *brief* description of the components and algorithms implemented in order to achieve a solution to the problem.

3.2 Configuration of computational dependencies

Before exposing the methodology and algorithms used to develop the application, we devote this section to describe the builds of the libraries that were used as dependencies for the application and it's configuration. For compilation and linking the final application we use `Makefile`s to obtain the application executable.

3.2.1 Boost

Boost[2] is a *header-only library*, which means that there's really nothing to build. It consists of header files containing templates and inline functions and require no separately-compiled library binaries or special treatment when linking.
This means that using Boost is as simple as including the header files in the source code and providing the directory of the header files at compilation time. We use `Makefile`s for compiling and linking the application, so, an environment variable can be declared here as `BOOST_DIR=/path/to/boost` and at compilation time we pass `-I $(BOOST_DIR)`.

3.2.2 PETSc

Building and installing a fully functional implementation of PETSc can provide a bit of a challenge. The library is huge and has many components, one can think of building only the components of PETSc that are required for the application, however, we build the library using all the components available. This provides more versatility if one were to introduce more functionality from PETSc to the application.

To start with, before building the library, the configuration requires to declare the location in which the library will be installed in the form of environment variables:

- `PETSC_DIR=/path/to/desired/location/petsc`
- `PETSC_ARCH=desired_name_for_installation`

PETSc allows the users to have different builds of the library with different configurations, `PETSC_ARCH` is used to differentiate them from one another.
We used the following configuration options to build a compatible PETSc installation with the required functionality to solve the problem:

1. `--with-shared-libraries=1`
   - We use shared libraries to reduce the size of the executables and reduce compilation times
2. `--with-mpi=1`

- In order to use MPI in the application. This should be done with care and will change from one computational environment to another. In a supercomputer or cluster, for example, PETSc should be installed using one of the local implementations of MPI already present such as OpenMPI, MPICH or Intel MPI. Allowing PETSc to download and install an implementation of MPI will most likely result in low performance or run time errors. For our particular case, we built PETSc using Intel MPI

3. `--with-debugging=0`
   - Performance is best when running applications compiled with PETSc without debugging mode
4. `--with-scalar-type=complex`
   - This allows the usage of the `PetscScalar` types as complex datatypes
5. `--with-64-bit-ints --with-64-bit-indices`
   - 64-bit integer representations using PETSc's own datatypes
6. `--with-fortran-kernels=1`
   - PETSc can use Fortran kernels to provide better performance when operating complex datatypes
7. `--with-blas-lapack-dir=/path/to/blas/lapack/dir`
   - PETSc can download and install these libraries on its own, however, this has to be specified if another implementation is to be used, such as Intel MKL. We used Intel MKL for our build

3.2.3 SLEPc

The SLEPc configuration rests on the build that was used for PETSc, so the only requirement is to pass to the configuration the directory in which the installation will be done and should be set as an environment variable:

- `SLEPC_DIR=/path/to/desired/location/slepc`

3.3 Serial implementation

A serial implementation was developed as a starting point in order to have a base for later parallelization. Part of the methodology used in this implementation holds even in the parallel version, although this serial application was never used for production runs, therefore, linear algebra operations were not optimized or offloaded to well-performing libraries: this was done at a later stage in the parallel version. Most of the simulations done with this implementation were used in order to verify results with the later parallel version.

This version is a C++ implementation that relies on the popular library *Boost* [2], namely the `dynamic_bitset` and `uBlas` components.

3.3.1 Construction of the Hilbert space basis representation

When it comes to calculating the dimension of the Hilbert space, datatype overflowing can occur[3], so instead of computing the dimension using the combinatorial factor, we do a very simple modification in order to avoid this. Algorithm 3.1 shows the pseudocode used for a member function that returns the dimension of the Hilbert subspace without overflowing the standard 64-bit integer datatypes for a much larger set of possible values of L and N:

Algorithm 3.1 Computing basis size

```
1: function BasisSize(void)
2:     double size = 1.0
3:     for LLInt i = 1 to L − N do
4:         size ∗= static_cast<double> (i + N) / static_cast<double> (i)
5:     end for
6:     return floor(size + 0.5)
7: end function
```

This is an important value to keep handy as it gives the size required for the basis and leading dimension of the Hamiltonian matrix. Memory allocation is done with this value in mind.

The next step is to actually compute the integers that represent each of the states of the subspace. In order to do this we use a contiguous section of memory of the required size. One could use standard C++ containers for this, but for specific design reasons related to memory management in later stages, we allocate memory using the regular mechanism by means of the `new` command and use raw pointers to access and modify elements. As was described before, a *lexicographical next bit permutation* approach provided all the required needs to compute the integer representation of the basis. This was implemented as shown in Algorithm 3.2.

[3]The value of 21!, for instance, already overflows the `long int` (64-bit ints) representation

Algorithm 3.2 Next permutation of bits

```
1: procedure ConstructIntBasis(LLInt *int_basis)
2:     LLInt w                                        ▷ Next permutation of bits
3:     LLInt smallest = smallest_int()                ▷ Smallest int of the basis
4:     int_basis[0] = smallest
5:     for LLInt i = 1 to basis_size_ − 1 do
6:         LLInt t = (smallest | (smallest −1)) +1
7:         w = t | ((((t & -t / (smallest & -smallest)) >> 1) −1)
8:         int_basis[i] = w
9:         smallest = w
10:    end for
11: end procedure
```

In such a way, the `int_basis` container gets filled with each of the possible permutations in a sorted manner. The `smallest()` function is a very simple routine that computes the smallest integer value of the bit representation corresponding for any given L and N.

`Boost`'s component, `dynamic_bitset`, provides easy to use functionality to create binary objects out of each of the elements of this container and moving from a binary to integer representation. This functionality was used for the later construction of the Hamiltonian matrix.

3.3.2 Construction of the Hamiltonian matrix

Once the basis of the subspace has been computed, the construction of the Hamiltonian matrix can be achieved based on this object and a given Hamiltonian operator. For this purpose, we used **periodic boundary conditions** and the Hamiltonian operator shown in Eq (2.1), although modifications to the actual Hamiltonian operator can be easily added in the code.

Algorithm 3.3 Binary to integer

```
1: function BinarytoInt(boost::dynamic_bitset<> bs)
2:     LLInt integer = 0
3:     for LLInt i = 0 to L − 1 do
4:         if bs[i] == 1 then
5:             integer += 1ULL << i
6:         end if
7:     end for
8:     return integer
9: end function
```

Algorithm 3.4 shows a minimalistic approach to the construction of the Hamiltonian and provides a base description of what was implemented both in the serial and parallel versions. A rotation term is added in the `next_site*` variables in order to produce periodic boundary conditions. Bit swaps were done using `Boost`'s `dynamic_bitset` functionality. For the conversion of binary objects to integer values, a simple approach was used as shown in Algorithm 3.3

Algorithm 3.4 Construction of the Hamiltonian matrix

```
1: procedure HamMat(LLInt *int_basis, double V, double t)
2:     for state = 0 to basis_size_ − 1 do
3:         bs = integer_to_binary(int_basis[state])
4:         for site = 0 to L − 1 do
5:             bitset = bs
6:             if bitset[site] == 1 then
7:                 int next_site1 = (site + 1)%L        ▷ Periodic boundary condition
8:                 if bitset[next_site1] == 1 then
9:                     (...add V to hamiltonian at position [state, state])
10:                else
11:                    (...do a swap...)
12:                    (...turn this object into a integer...)
13:                    (...look for the integer in int_basis...)        ▷ Important!
14:                    index1 = position_in_basis(...)
15:                    (...add t to hamiltonian at position [index1, state])
16:                end if
17:            else
18:                int next_site0 = (site + 1)%L        ▷ Periodic boundary condition
19:                if bitset[next_site0] == 1 then
20:                    (...do a swap...)
21:                    (...turn this object into a integer...)
22:                    (...look for the integer in int_basis...)        ▷ Important!
23:                    index0 = position_in_basis(...)
24:                    (...add t to hamiltonian at position [index0, state])
25:                else
26:                    (...do nothing...)
27:                end if
28:            end if
29:        end for
30:    end for
31: end procedure
```

The lookup section of Algorithm 3.4 is indeed very important. One could do an *element-by-element* lookup for each of the values in the basis buffer and this would be satisfactory, if the size of the system is kept small. However, for the study of

larger systems a better way is required. As per the description before, we decided to sequentially compute the values of the basis by means of bit permutations, in such a way that the next permutation of a given `bitset` corresponds to the binary combination that provides the next integer value when translated into an integer representation. One of the benefits of doing this is not only it's performance, but the fact that the resulting integer representation of the basis is **sorted**.

In light of this, we can use a well-performing method to lookup entries of the memory buffer. For this particular case[4], a very good choice is to use a **binary lookup** and therefore using an algorithm with complexity $O(logN)$ instead of an element-by-element lookup with complexity $O(N)$. Both in the serial and parallel versions, this accounts for a good optimization when it comes to the computational walltime of this section of the code. Algorithm 3.5 shows the lookup method that was implemented in the application.

Algorithm 3.5 Binary lookup

```
1: function BinSearch(const LLInt *array, LLInt len, LLInt value)
2:     if len == 0 then
3:         return -1
4:     end if
5:     LLInt mid = len / 2
6:     if array[mid] == value then
7:         return mid
8:     else if array[mid] < value then
9:         LLInt result = BinSearch(array + mid + 1, len - (mid + 1), value)
10:        if result == −1 then
11:            return -1
12:        else
13:            return result + mid + 1
14:        end if
15:    else
16:        return BinSearch(array, mid, value)
17:    end if
18: end function
```

3.3.3 Time evolution: Krylov subspace methods approach

The methodology and approach used in this section relies on the development presented by [9]. Both the numerical approach and implementation are developed and described in this reference. The original work presented there lacks parallelization,

[4]A sorted array of integer values

which is a requirement (in terms of time to solution and memory) to solve the problem at hand in an efficient manner for larger sizes.
The serial implementation developed differs only in the programming language and the libraries used to accomplish the numerical operations and was used to study the methodology and approach, as well as to test the results produced from the numerical computations on the later developed parallel version.
We leave the discussion involving the implementation of this, additional details, performance and simulation results to the parallel version section of this document.

3.4 Parallel version

In this section we describe the implementation details of the parallel approach used to solve the problem. This development constitutes the biggest part of the effort devoted to solving the problem efficiently.
Parallelization in the sense of the problem described not only allows the usage computational resources efficiently distributed in order to reduce time to solution, but it also translates into the possibility to solve *much larger systems that are unsolvable from the computational point of view otherwise.*

Several different parallel implementations were developed[5], however, we expose here four different approaches used to solve a specific problem: large memory management and it's relation to computation time.
Unlike the serial version, this implementation is designed to be used in production runs and most of our results were obtained using an application developed with the following descriptions.

This version is a C++ application that relies on *Boost* [2] for binary operations and some mathematical functions, and both PETSc and SLEPc [1][11] for the matrix objects and time evolution. Part of the development is designed to be compatible with PETSc, such as the parallel distribution of the Hamiltonian matrix or time evolution vectors, for instance. More about parallel distribution in PETSc and the way the matrix objects are constructed can be consulted in [1].
The following section provides a brief description of the methods used towards developing a parallel application.

3.4.1 Design

In Figure 3.2 we provide a brief description of the parallel design.
One of the biggest problems related to the unitary time evolution of these systems is related to amount of memory that is required to solve the problem efficiently, this not a problem for small systems sizes, but it grows with a very rapid rate with increasing L.

A drawback of this particular approach is the fact that the Hamiltonian matrix needs to be stored in memory (in a sparse format, to reduce memory consumption) and that the basis needs to be accessed and stored in particular ways in order to construct this matrix efficiently. Part of the design is, in this sense, to develop a mechanism for which the construction of the Hamiltonian matrix step can be done and that this procedure scales with the number of computation elements.

[5]Differences with each other differ mainly in the construction of the Hamiltonian section of the problem

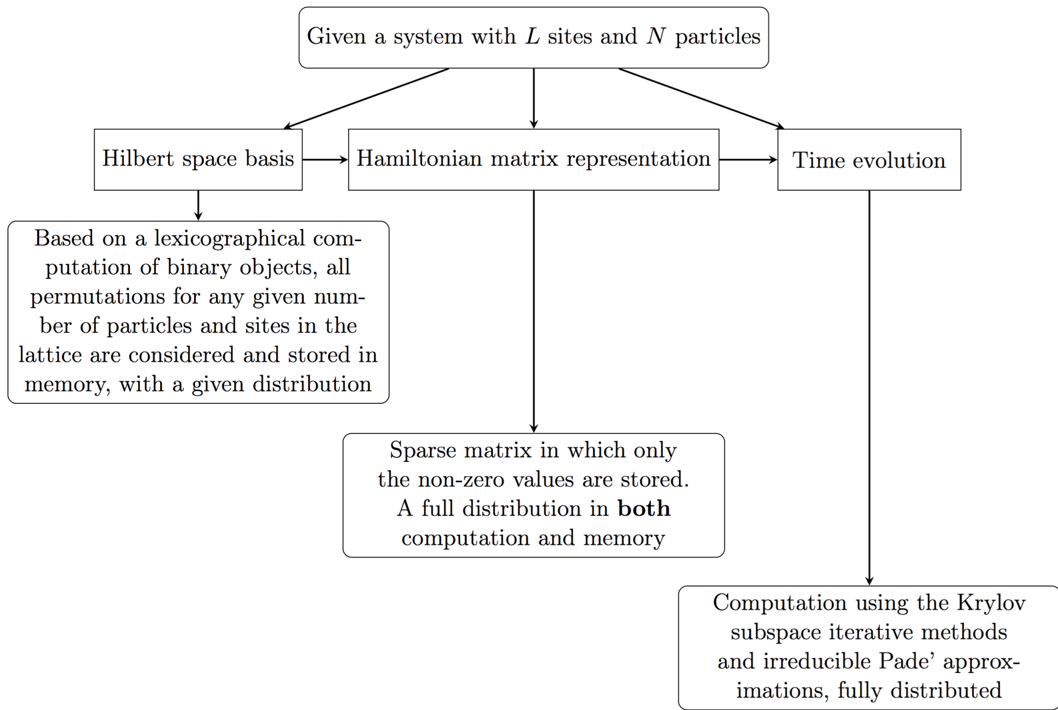

Figure 3.2: Brief summary of the parallel design

Table I. Memory consumption estimation for different problem sizes at *half-filling*.

System sizes	Problem size	Basis memory (GB, exact)	Matrix memory (GB, overestimation)
$L = 28,\ N = 14$	40 116 600	0.320	18
$L = 30,\ N = 15$	155 117 520	1.25	75
$L = 32,\ N = 16$	601 080 390	4.8	308
$L = 34,\ N = 17$	2 333 606 220	18.7	1269
$L = 36,\ N = 18$	9 075 135 300	72.6	5 227
$L = 38,\ N = 19$	35 345 263 800	282.8	21 490

Table I presents a numerical estimation of memory consumption in order to illustrate the previously described drawback. In the estimation presented in Table I, there's no estimation in the amount of resources (memory) to achieve the *actual time evolution*. For instance, the method requires to obtain projections on the Krylov subspace, which means that the actual memory consumption for the entire application is in fact larger. Leaving this fact aside for the moment, Table I provides an insight on how the dimension of the problem is related to computing resources.

We have measured, using PETSc's own profiler tool, that the actual overall memory requirement to perform the simulations including time evolution can go as far as **4**

times the estimated values for the matrix, this is accounted by internal memory allocation required to perform the time evolution procedure.

Estimation was produced as follows: the memory required for one buffer containing the basis is equal to dimension of the subspace, or *problem size*, times the size in bytes for each of the objects. In this case this corresponds to 64-bit integers, if one were to use 32-bit representations the datatype would overflow at certain point for a given size. For the case of the Hamiltonian matrix, the estimation counts the dimension of the subspace times the size in bytes of the `double precision complex` datatype. An additional factor is introduced in this estimation to account for the sparsity of the matrix, roughly speaking, an overestimation would be computed as follows:

$$GB_{matrix} \sim 16\ Bytes \times \frac{L!}{N!(L-N)!} \times L \tag{3.1}$$

Which is far less amount of memory in comparison to a dense representation of the matrix.
The information provided in Table I suggests that at some point, partial or full distribution of the basis is going to be required in order to solve larger systems without exhausting intranode memory resources. The Hamiltonian matrix also requires a full distribution across nodes.
Another fact that can be extracted from Table I is that up to a certain problem size the unitary time evolution can't be computed using a single computational node, hence, a full distribution using *Message Passing Interface* is required.

In the following subsections we provide a description of four particular versions used to tackle the problem with the previously described considerations in mind.

3.4.2 Replicated basis version

This constitutes the first instance towards a parallel application. It can be seen from Table I that for a large set of problem sizes[6], basis replication isn't really a problem to be concerned with memory-wise. Basis replication in this context means having a memory section devoted to contain the integer values representing each of the states in the subspace per each computing element, hence, there's no distribution on the basis memory across processing elements. In *Message Passing Interface* terms, each MPI process allocates and has access to the memory address of the entire basis. However, the Hamiltonian matrix object and the time evolution is performed with full distribution across processing elements in this version.

[6] $L = 28$ at half filling and smaller values of L, plus all the systems with subspace dimension smaller than this one

Even though we kept the memory requirements shown in Table I in mind during development, this first version allowed us to analyze the overall performance for moderate system sizes in which basis replication is not a problem. With this version we introduced the functionality of PETSc and SLEPc for the first time into the application, so this allowed us also to study the behavior of the library and a base line for further distribution of the basis and development.

We devote this section to describe the changes involving the main parts of the implementation in light of what has been described in the serial version section.

Construction of the Hilbert space representation

Given that for this version replication of the basis is allowed, there's no change involving the computation of the basis. The computation is done in the same manner as described in Section 3.3.1.

In the sense of Object Oriented programming, paradigm that was used to implement the application, each MPI process creates an instance of the `class Basis` and each process manages this resource on it's own; such as reclaiming the memory resources at the end of the application.

Construction of the Hamiltonian matrix

As stated in Figure 3.2, we intend to make this section of the application run with full distribution in both memory resources and computation of matrix elements. In this sense, a mechanism to assign distribution is required.
In order to make this distribution compatible with the internal MPI distribution that PETSc uses for it's objects, we use the same row distribution per computing elements, as shown in Algorithm 3.6

Algorithm 3.6 Parallel distribution

1: **procedure** DISTRIBUTION(PetscInt &$nlocal$, PetscInt &$start$, PetscInt &end)
2: $\quad nlocal = basis_size_ \,/\, mpisize_$
3: $\quad$ PetscInt $rest = basis_size_ \,\%\, mpisize_$
4: $\quad$ **if** $rest$ && $(mpirank_ < rest)$ **then**
5: $\qquad nlocal++$
6: $\quad$ **end if**
7: $\quad start = mpirank_ * nlocal$
8: $\quad$ **if** $rest$ && $(mpirank_ >= rest)$ **then**
9: $\qquad start += rest$
10: $\quad$ **end if**
11: $\quad end = start + nlocal$
12: **end procedure**

The use of a particular datatype can be noticed: `PetscInt`. Using the configuration shown in Section 3.2 this makes compatibility easier with the routines from PETSc. The variables `mpirank_` and `mpisize_` are queried with the usual MPI directives and taken as private members of the `class Hamiltonian`. The constructor of this `class` initializes the PETSc, SLEPc and MPI environments. The variables `nlocal`, `start` and `end` provide global indices and sections of the `row` distribution. This is a very standard parallel row distribution across processing elements and most of the PETSc routines stick to this distribution.
The next step is to construct the matrix representation of the Hamiltonian operator in parallel. For this purpose we use a PETSc `Mat` object in `MATMPIAIJ` format. Each processing element or MPI rank will contain in memory only it's own section of the matrix given by `Distribution` and compute the elements of the same section. Collective operations are done at a later stage through MPI communication.

Algorithm 3.7 shows the mechanism used to construct and assemble the Hamiltonian matrix parallel object. As can be seen, for this version the algorithm is very similar to the serial version, however, there are important differences worth mentioning. The whole procedure is *row subdivided*, which means that each MPI process will only have direct access to it's own section of memory addresses for the matrix object and computation of elements is also distributed, which means we expect this procedure to scale as the number of processing elements increases.
The first procedure of Algorithm 3.7 is important. There are basically three different ways in which a PETSc `Mat` object can be constructed:

1. Create an instance of the object without specifying preallocation
2. Create an instance of the object providing estimated values of sizes for preallocation
3. Create an instance of the object providing the exact amount of elements in the diagonal and off-diagonal portions of the matrix with the parallel subdivision taken into account

Out of the three methods, the first one is the simplest but performs the worst. This is because of the *overhead* related to dynamically resizing memory sections. The second method performs well if a good estimation is provided, this usually requires allocating more memory than what it's actually required for the object.

Given that our goal is to optimize memory consumption, the third method described above is the best for our purposes. This requires implementing another routine similar to the one shown in Algorithm 3.7, that computes the elements that each processing element contains in it's own section of the matrix, so that a **preallocation** step can be performed. This can also be performed in parallel in order to avoid compromising scalability.

Algorithm 3.7 Parallel Construction of the Hamiltonian matrix

```
1: procedure RepHamMat(LLInt *int_basis, double V, double t, PetscInt
   nlocal, PetscInt start, PetscInt end)
2:     determine_allocation_details_(...)                        ▷ Important!
3:     (...instantiate the Mat object with the allocation details...)
4:     for state = start to end − 1 do
5:         bs = integer_to_binary(int_basis[state])
6:         for site = 0 to L − 1 do
7:             bitset = bs
8:             if bitset[site] == 1 then
9:                 int next_site1 = (site + 1)%L     ▷ Periodic boundary condition
10:                if bitset[next_site1] == 1 then
11:                    (...add V to hamiltonian at position [state, state])
12:                else
13:                    (...do a swap...)
14:                    (...turn this object into a integer...)
15:                    (...look for the integer in int_basis...)  ▷ Binary lookup routine
16:                    index1 = position_in_basis(...)
17:                    (...add t to hamiltonian at global position [index1, state])
18:                end if
19:            else
20:                int next_site0 = (site + 1)%L     ▷ Periodic boundary condition
21:                if bitset[next_site0] == 1 then
22:                    (...do a swap...)
23:                    (...turn this object into a integer...)
24:                    (...look for the integer in int_basis...)  ▷ Binary lookup routine
25:                    index0 = position_in_basis(...)
26:                    (...add t to hamiltonian at global position [index0, state])
27:                else
28:                    (...do nothing...)
29:                end if
30:            end if
31:        end for
32:    end for
33:    (...begin assembling procedure...)
34:    (...end assembling procedure...)
35:    (...indicate PETSc that this is a symmetric matrix...)
36: end procedure
```

This introduces computational overhead, but the mechanism performs well enough so that we can use this in order to save memory resources.
The mechanism used is very similar to the one shown in Algorithm 3.7, but instead of adding elements to the matrix object in lines `11, 17` and `26` we count the number of elements in the diagonal and off-diagonal subsection of the matrix owned by each MPI process as shown in Algorithm 3.8.
The steps involving the PETSc `Mat` object are not applied here, since this procedure only computes the number of elements so a proper, well-performing preallocation step can be done.

Algorithm 3.8 Parallel allocation details

```
 1: procedure DetermineAllocationDetails_(...)
    ...
17:     if index1 < end && index1 >= start then
18:         diagonal[state − start]++
19:     else
20:         off[state − start]++
21:     end if
    ...
22: end procedure
```

Time evolution

Now that the matrix representation has been constructed in a distributed fashion and in consistency with the parallel distribution required to the PETSc and SLEPc routines, we can proceed to evaluate the time evolution of the system with an initial state. The initial state can be constructed in many different ways to study different behaviors of the system and should be represented as a vector distributed in parallel among processing elements, this can be easily done using PETSc and the given distribution shown in Algorithm 3.6. For the time evolution procedure we can use SLEPc's routine related to the MFN component, which provides all the necessary framework with enough versatility to carry out the computation. Algorithm 3.9 shows a minimalistic approach to implementing this, we will not focus on the APIs or actual function calls, this can be consulted in [11].
Algorithm 3.9 shows just an example in which the time evolution can be performed, though this will change depending on what properties of the system are to be evaluated and computed observables. PETSc and it's extension SLEPc are very complete libraries with a lot of versatility implemented. We have performed and evaluated time evolution using Krylov subspace methods for our own purposes, but the other components can be used on the constructed Hamiltonian in order to, for example, do exact diagonalization or perform different mathematical procedures.

Algorithm 3.9 Time evolution

```
 1: procedure TimeEvolution(...)
 2:     (...create and change parameters of the MFN component...)
 3:     (...declare and allocate a helping vector for time evolution...)      ▷ Can be
    avoided by using in place directives
 4:     for tt = 1 to iterations + 1 do            ▷ tt is the number of time steps
 5:         (...scale the FN component by the current tt...)
 6:         MFNSolve(...)
 7:         (...get converging reason and abort if convergence is not achieved...)
 8:         (...check that the time-evolved vector retains norm...)        ▷ Optional
 9:         (...evaluate the time-evolved vector in any desired way...)
10:         (...evolved vector becomes new initial vector...)
11:     end for
12:     (...destroy helping vector...)
13:     (...destroy MFN environment...)
14: end procedure
```

3.4.3 Node communicator version

Taking a closer look at Table I, we can see that replicating the basis per computing element will rapidly exhaust memory resources, particularly if a large set of MPI processes are used to perform the computation.
This has to be dealt with if one is interested in studying large system sizes, or in this case, systems with a large subspace dimension.

We focus our attention now on the first method that was used to overcome this problem. The paradigm consists in distributing the basis among all the processing elements, except for the first MPI process of each node, which allocates and holds the memory addresses of the entire basis. In this scenario, the entire memory required for the basis alone would be: 1 entire basis per node plus 1 entire basis distributed across the rest of the MPI processes. Computations required to construct the Hamiltonian matrix then require intranode communications to find missing information. One of the benefits that are posed by this solution is the fact that the communication is being done inside of the node, most MPI implementations benefit from this using hardware locality directives. Figure 3.3 shows a visual representation of this.
We want to use this particular distribution for the construction of the Hamiltonian, however, the time evolution computation should use the distribution shown in the previous subsection using the global communicator since that provides the best balance and compatibility with PETSc functionality.
This can be done by means of a second MPI communicator. We called this second communicator `node_comm_` and it's a private member of the class `Hamiltonian`. As of the release of the MPI 3.0 standard there's a very natural way to accomplish this task by means of the MPI shared regions:

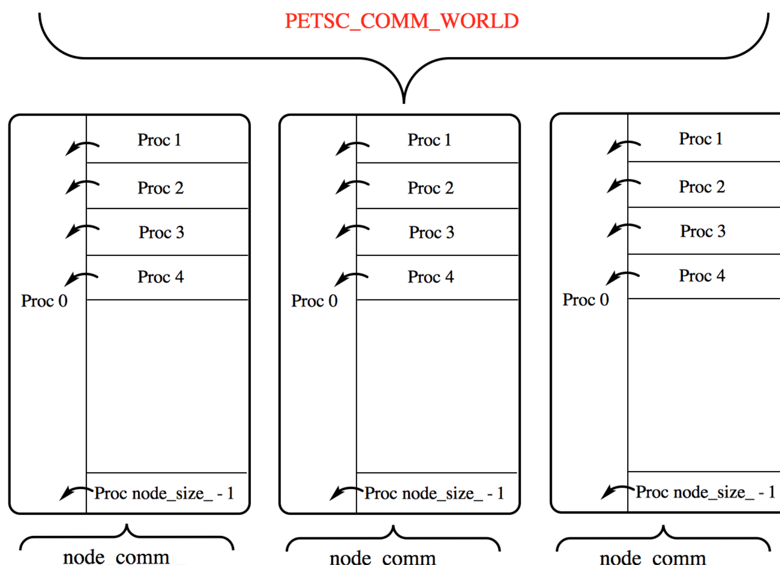

Figure 3.3: Visual representation of the node communicator version

```
...
MPI_Comm node_comm_;
MPI_Comm_split_type(PETSC_COMM_WORLD, MPI_COMM_TYPE_SHARED,
  mpirank_, MPI_INFO_NULL, &node_comm_);
...
```

The `node_comm_` MPI communicator can be used to establish communication patterns of computing elements within a local node. Given that the first MPI process of each node contains the entire basis, communication can be performed in order to construct the matrix representation of the Hamiltonian operator using the data contained by this computing element.

Construction of the Hilbert space representation

Given that the basis is distributed in a different fashion in relation to the replicated basis version, the parallel paradigm to compute the basis needs to be changed. In particular, each MPI process will allocate and hold the addresses in memory of only a section of the basis with the exception of the first process of each node as described before. Using the same distribution shown in Algorithm 3.6, we can redesign the `public method` of the `class Basis` that computes the basis of the system in order to introduce the parallel distribution. This is shown in Algorithm 3.10.
It can be noticed that each MPI process will compute only it's own section of the basis. The `int_basis` buffer has size *nlocal*[7] given by the distribution in Algorithm 3.6. The method `first_int()` is a very simple routine that computes the first

[7] $nlocal = basis_size_$ for the first MPI process of every node

Algorithm 3.10 Construction of the basis in the Node communicator version

```
1: procedure ConstructIntBasis(LLInt *int_basis, PetscInt nlocal, PetscInt start)
2:     LLInt w                                        ▷ Next permutation of bits
3:     LLInt first = first_int(nlocal, start)   ▷ Smallest int of the section of the basis
4:     int_basis[0] = first
5:     for LLInt i = 1 to nlocal − 1 do
6:         LLInt t = (first | (first −1)) +1
7:         w = t | ((((t & -t / (first & -first)) >> 1) −1)
8:         int_basis[i] = w
9:         first = w
10:    end for
11: end procedure
```

element in the basis for a given MPI process.

Construction of the Hamiltonian matrix

In relation to what was described in the previous design using a replicated basis, we use the new communicator both in the `DetermineAllocationDetails` and in the `HamMat` procedures for the purposes of constructing the Hamiltonian matrix. In this particular scenario, each MPI process will take ownership of a given subsection of rows in the matrix and compute the elements of the matrix by means of it's own basis section. Some elements of the matrix won't be able to be computed given that the local basis is incomplete, so this information is stored and communicated to the first MPI process of the local node in order to complete the procedure.

Algorithm 3.11 shows the first section of the `DetermineAllocationDetails` routine, most of the changes in relation to the replicated basis version are in the indexing of the elements with this new distribution but there's a new function at the end of the procedure: `NodeComm`. This routine uses MPI directives to perform the communication required in order to complete the construction of the Hamiltonian matrix.

At this point, each of the processes that are not the first MPI process have computed integer representations of states that are not found in the basis, given that they have access to only a portion of the basis. So what we do is use the `NodeComm` routine to find missing information that can be queried from the first MPI process of the local node. There's a caveat to this approach: looking at Figure 3.3, the communication pattern has to be ordered in a sequential fashion; i.e, each process of the node communicates to the first process of the node one after the other. This has to be done in order to avoid exhausting of the local node memory resources. In spite of this, we have found that scalability of this section of the implementation is

Algorithm 3.11 Node communicator parallel allocation details of the Hamiltonian matrix

```
 1: procedure DetermineAllocationDetails(...)
 2:     (...diag[0 : nlocal] = 1...)
 3:     for state = start to end − 1 do
 4:         PetscInt basis_ind;
 5:         node_rank_ ? basis_ind = state − start : basis_ind = state;
 6:         bs = integer_to_binary(int_basis[basis_ind])
 7:         bool counter = false;
 8:         for site = 0 to L − 1 do
 9:             bitset = bs
10:             if bitset[site] == 1 then
11:                 int next_site1 = (site + 1)%L          ▷ Periodic boundary condition
12:                 if bitset[next_site1] == 1 then
13:                     counter = true;
14:                 else
15:                     (...do a swap...)
16:                     (...turn this object into a integer...)
17:                     (...look for the integer in int_basis...)  ▷ Binary lookup routine
18:                     LLInt index1;
19:                     if node_rank_ then
20:                         index1 = position_in_basis(...)
21:                         (...if not found store the unfound value, otherwise keep it...)
22:                     else
23:                         index1 = position_in_basis(...)                ▷ Rank 0 of node
24:                     end if
25:                     if index1 < end && index1 >= start then
26:                         diagonal[state − start]++
27:                     else
28:                         off[state − start]++
29:                     end if
30:                 end if
31:             else
32:                 ...
33:             end if
34:         end for
35:         (...reduce diagonal by 1 if counter is false...)
36:     end for
37:     NodeComm((...))
38: end procedure
```

Algorithm 3.12 Node communication pattern

```
1: procedure NodeComm(...)
2:     // Communication to rank 0 of every node to find size
3:     if node_rank_ then
4:         MPI_Send(...)
5:     else
6:         for i = 1 to node_size_ do
7:             MPI_Recv(...)
8:         end for
9:     end if
10:    // Communication to rank 0 of node to find missing indices
11:    if node_rank_ then
12:        MPI_Send(...missing info...)              ▷ Send unfound information
13:        MPI_Recv(...required info...)    ▷ Required to finish the construction
14:    else                                            ▷ Rank 0 of every node
15:        for i = 1 to node_size_ do
16:            MPI_Recv(...missing info of rank i)
17:            (...binary search...)
18:            MPI_Send(...required info to rank i)
19:        end for
20:    end if
21:    (...complete construction with the updated data...)
22: end procedure
```

not compromised and the construction of the Hamiltonian matrix can be performed in a timely manner in relation to the previous version, which was exactly our goal.

Furthermore, Algorithm 3.12 shows the MPI approach implemented to accomplish this.

In this scenario, the first step would be to communicate the amount of elements that are going to be sent from each process to the corresponding *rank 0* of every node, afterwards the actual communication and binary search is performed on every *rank 0* of every node to find the required data to finish the construction of the Hamiltonian matrix before communicating again the updated data back to the sender.

After the computation of the allocation details is completed, a routine that fills the elements of the distributed matrix needs to be called. The routine is very similar to the one shown in Algorithm 3.11 but instead of counting diagonal and off-diagonal elements, the values are introduced into the matrix. This happens in lines 25-29 in Algorithm 3.11. All the missing information resulting from the distribution of the basis has been updated at this point so the communication pattern is no longer required and the matrix can be filled from each of the processes using these data

segments.

We have found that this mechanism provides a good performing solution to the replicated basis problem. Performance and results can be found in the next chapter of this document.

Before performing the time evolution of the system, the basis memory is reclaimed and the communicator used to perform this task is switched back to the global communicator, or `PETSC_COMM_WORLD`, as shown in Figure 3.3. In this sense, there's no modification required to the time evolution step.

3.4.4 Ring exchange version

Depending on the available memory resources per node of the computational environment in which the application is executed, according to the estimates shown in Table I, allocating and constructing one entire basis per node can exhaust the memory resources of the system for large system sizes. Even if this could be done, there's still memory resources required for the actual Hamiltonian matrix. The node communicator version provides a good solution for a very large range of system sizes, however, for systems that have a very large subspace dimension a fully distributed approach needs to be devised.

We developed an algorithm that uses a ring exchange of the subdivided basis among all the processing elements. In relation to our previous `NodeComm` setup, this version differs in the following:

- Full distribution of the basis across all the processing elements
- Unlike the previous version, communication is done in order to exchange sections of the basis and not the *unfound* elements of the Hamiltonian matrix. This is an important difference: computational load is more balanced in this setup
- The algorithm can be implemented by means of a single MPI communicator

In this new setup then, each processing element will exchange sections of the basis in order to compute the elements of the Hamiltonian matrix. After $mpi_processes - 1$ exchanges, all the elements have been computed and the Hamiltonian matrix is computed and distributed. With this setup, a *linear scaling* in memory is achieved which means that with increasing number of processing elements the amount of memory per MPI process required decreases linearly. However, *time scaling* is compromised, as increasing the number of processing elements will require more communication steps. Therefore, we expect that with this setup the time required to compute the Hamiltonian matrix will increase in relation to previous procedures.
Results obtained with this setup are explained in greater detail in the next chapter.

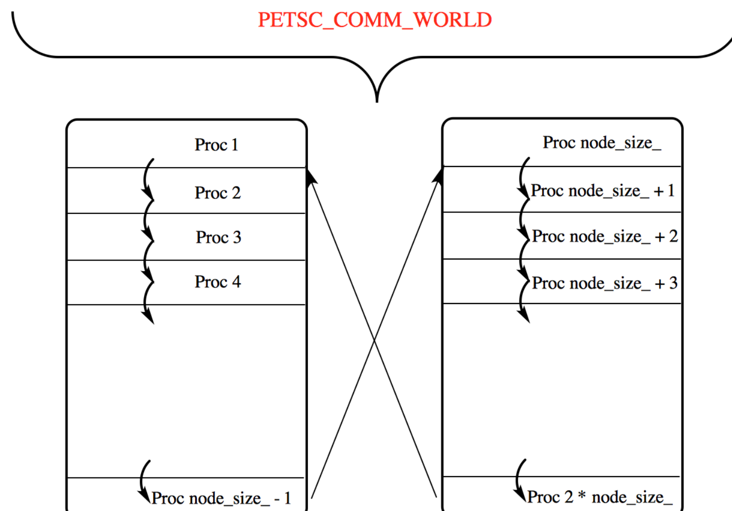

Figure 3.4: Visual representation of the ring exchange version

Construction of the Hilbert space representation

The same procedure used in Algorithm 3.10 can be used here, each processing element will compute it's section of the basis of size `nlocal` by means of the same method of the `class Basis`.

Construction of the Hamiltonian matrix

In relation to the previous description used for the `NodeComm` setup, the major difference is given by the communication step. There needs to be some modification regarding the indexing of elements from the procedure shown in Algorithm 3.11, but the changes are not the essence of the algorithm so we will focus on the communication step instead.
Algorithm 3.13 shows the procedure used to perform the ring exchange of the basis in order to compute the Hamiltonian matrix. The first step is to to perform a collective communication step so that every processing element holds the *nonlocal* values of the indices of each subsection of basis being communicated. This has to be done to keep track of the positions of the elements in the Hamiltonian matrix. Afterwards, since the number of processors is a generic parameter, different MPI processes may hold larger sections of the basis than others. To account for this, we use the larger size for the exchanging buffers with the remaining elements set to zero, as this won't affect the binary lookup given that no state in the basis is represented by the zero value.
The variable `source` identifies the MPI process that sends the section of the basis, this is required in order to align the indices with global values. The last step is to perform the ring exchange and the binary lookup procedure until all sections of the

Algorithm 3.13 Ring communication pattern

```
 1: procedure RingComm(...)
 2:     // Collective communication of global indices
 3:     gather_nonlocal_values_(...);
 4:     // Even when rest != 0, Proc 0 always gets the larger section
 5:     // of the distribution, so we use this value for the ring exchange buffers
 6:     broadcast_size_of_buffers_(...);                         ▷ From 0 to all
 7:     (...allocate ring exchange buffers, initialized to basis in a sorted fashion...)
 8:     PetscMPIInt next = ( mpirank_ + 1 ) % mpisize_;
 9:     PetscMPIInt prec = ( mpirank_ + mpisize_ - 1 ) % mpisize_;
10:     for PetscMPIInt exc = 0 to mpisize_ − 2 do
11:         MPI_Sendrecv_replace_(...);       ▷ Basis exchange using prec and next
12:         // source is required to find global indices
13:         // from nonlocal values
14:         PetscMPIInt source = mod_((prec - exc), mpisize_);
15:         (...binary lookup of elements, if found, assign and multiply by −1)...
16:     end for
17:     (...flip all the signs: multiply by -1 all elements...)
18: end procedure
```

basis have been exchanged. In order to keep track of the already found elements, we set then to negative values and when the procedure is done we flip them back to positive entries. At the end of the procedure all the elements of the Hamiltonian matrix have been found and computed.

For the following step, which is to actually insert the elements to Hamiltonian matrix, we can use the already computed values and the ring exchange is no longer required. Furthermore, the time evolution of the system; as in the previous version, is left unchanged by this mechanism so we use the same procedure shown in Algorithm 3.9.

3.4.5 Combined Ring-Node communicator version

The ring exchange version presented before constitutes the best scenario when it comes to memory scalability: no sections of memory are replicated and everything is computed in a fully distributed fashion in all sections of the implementation. There's no memory replication whatsoever.
The caveat of this implementation is related to the *strong scalability* of the application, given that increasing the number of processing elements increases the amount of communications overall. This is not a concerning problem given that the computation of the Hamiltonian matrix is not a computationally intensive task, so even in the ring exchange design the time required to construct the Hamiltonian matrix object is small compared to the time evolution procedure. On an effort to recover

strong scalability of the application we have developed a final version that combines the two previous descriptions: node communication and ring exchanges.

The Node communicator version described in Section 3.4.3 relies on distributing the entire basis across all processing elements, with the exception of the first MPI process of each node which constructs and holds in memory the entire basis. This new combined version is similar to some extend: we distribute the entire basis across all MPI processes, but instead of allocating the entire basis on the first MPI process of each node we only construct and allocate another subdivision of the basis based on the number of computing nodes used. In this scenario a first step in constructing the Hamiltonian matrix for every MPI process is to communicate to the first MPI process of every node to find values connected to nonlocal sections of the basis, afterwards a ring exchange is done between each `rank 0` of every node until a full cycle is done. Figure 3.5 shows a visual representation of this description.

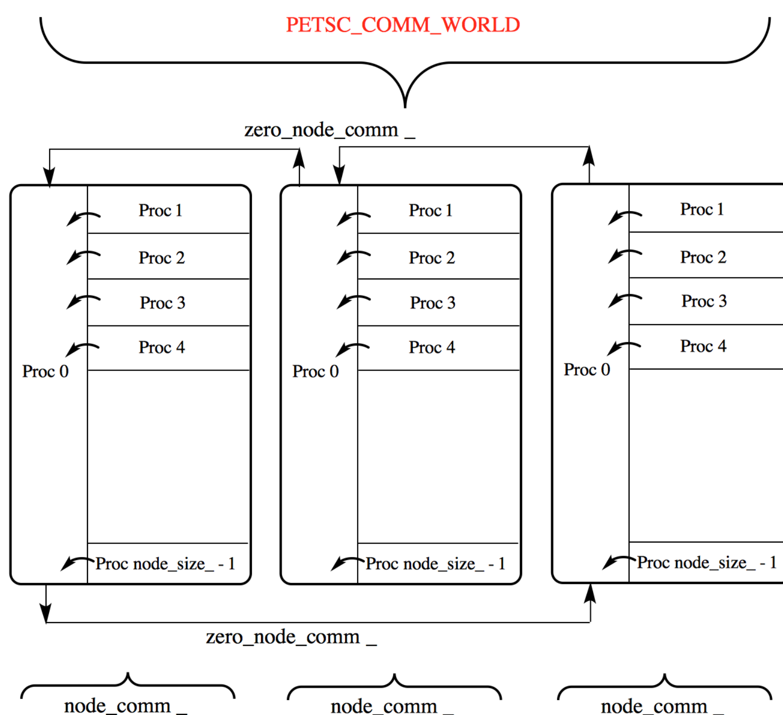

Figure 3.5: Visual representation of the combined Ring-Node communicator version

At this point two different MPI communicators have been used to solve the problem: `PETSC_COMM_WORLD` which involves all the processing elements and `node_comm_` which groups all the MPI processes of a given node.
The current task can be implemented by means of a third communicator: `zero_node_comm_`. This is more of a communicator group in the MPI language. Algorithm 3.14 shows the approach used to accomplish this.

Once the communicator has been established, it can be used to perform collective operations between the MPI processes of the group defined. In our particular case, a ring exchange procedure will be done by means of the MPI processes that are part of this communicator.

Algorithm 3.14 Establishing a communicator group: `zero_node_comm_`

```
1: procedure RingNodeComm(...)
2:     MPI_Group petsc_group;
3:     MPI_Comm_group(PETSC_COMM_WORLD, &petsc_group);
4:     PetscInt nodes = (( mpisize_ - 1 ) / node_size_ ) + 1;
5:     int *ranks = new int[nodes];
6:     for i = 0 to nodes - 1 do
7:         ranks[i] = node_size_ * i;
8:     end for
9:     MPI_Group zero_of_node_group;
10:    MPI_Group_incl(petsc_group, nodes, ranks, &zero_of_node_group);
11:    MPI_Comm zero_node_comm_;
12:    MPI_Comm_create_group(PETSC_COMM_WORLD,
   zero_of_node_group, 0, &zero_of_node_group);
   ...
   (...rest of communication procedure goes here...)
   ...
13:    delete [] ranks;
14:    MPI_Group_free(&petsc_group);
15:    MPI_Group_free(&zero_of_node_group);
16:    MPI_Comm_free(&zero_node_comm_)
17: end procedure
```

Construction of the Hilbert space representation

The public methods described in previous sections contained in the `class Basis` to construct the basis are still valid in this scenario but since the distribution changes from the processors that are part of the `zero_node_comm_` to the ones that are not, a new distribution method is to be used for the first MPI processes of each node. For this distribution we can use a similar approach to the one used all along: a row distribution of each of the subdivisions of the basis for the MPI processes in the `zero_node_comm_`. Algorithm 3.15 shows a way to accomplish this, the method is called from all the processors so for those that are not part of `zero_node_comm_` we can set the descriptors to -1 just to avoid confusion. In order to construct and allocate the basis for these MPI processes, one only has to call the same method described in Algorithm 3.10 using these new parameters.

Algorithm 3.15 Parallel distribution

```
1: procedure NodeDistribution(PetscInt &nlocal, PetscInt &start, PetscInt &end)
2:     nlocal = -1; start = -1; end = -1;
3:     if node_rank_ == 0 then
4:         PetscInt nodes = (( mpisize_ - 1 ) / node_size_ ) + 1;
5:         PetscInt node_0_id = mpirank_ / node_size_;
6:         nlocal = basis_size_ / nodes;
7:         PetscInt rest = basis_size_ % nodes;
8:         if rest && (node_0_id < rest) then
9:             nlocal + +
10:        end if
11:        start = node_0_id * nlocal;
12:        if rest && (node_0_id >= rest) then
13:            start += rest
14:        end if
15:        end = start + nlocal;
16:    end if
17: end procedure
```

Construction of the Hamiltonian matrix

Now that the basis has been computed with the desired distribution, the Hamiltonian matrix can be constructed by means of this object. Once again, the new parallel distribution of elements implicates that the way the elements of the Hamiltonian matrix in relation to the elements of the basis needs to be done somewhat differently. Simply put, the relationship between global and local indices changes depending on the MPI process that computes the elements of the matrix. This is a simple modification so we will focus on the most important change instead: the communication procedure.

In this particular case the procedure is a combination of the methods exposed before. We start by performing communications from each of the MPI processes to the first MPI process of the local node to compute the unfound matrix elements due to incomplete basis. However in this scenario, the local node *masters* also hold an incomplete basis so in order to compute the elements a ring exchange is done by means of `zero_node_comm_` as described in Section 3.4.4. This methodology is design specifically for very large physical system sizes and there's an important caveat to using this approach: given the large amount of elements to be computed, the intranode communication procedure has to be done in a sequential fashion in order to use the same section of memory for all processes. If one were to communicate all the elements in a single step, because of the size of the system, memory would exhaust very quickly. In this sense, the intranode communication procedure has to be done one MPI process after the other. Furthermore, this intranode communication step

has to be done after each ring exchange cycle for the same reason. The performance section of the next chapter will provide more insight regarding the approaches used to solve the problem of large memory requirements.
Algorithm 3.16 exposes the communication pattern approach used in this version.

The procedure shown in Algorithm 3.16 is called during the computation of allocation details of the parallel matrix. Like in the previous descriptions, the next step is to actually fill the elements of the matrix by means of PETSc directives but in this step the communication pattern shown before is no longer required and the matrix can be constructed by means of the already computed elements. Likewise, the time evolution procedure is unaffected by this treatment so we proceed with the same mechanism shown in Algorithm 3.9

Algorithm 3.16 Ring-Node communication pattern

```
 1: procedure RingNodeComm(...)
 2:     (...communicator object construction shown in Algorithm 3.14...)
    ...
 3:     // Collective communication of global indices, only ranks 0 of node
 4:     gather_nonlocal_values_(...);            ▷ By means of MPI_Allgather on
    zero_node_comm_
 5:     // Even when rest != 0, Proc 0 always gets the larger section
 6:     // of the distribution, so we use this value for the ring exchange buffers
 7:     broadcast_size_of_buffers_(...);         ▷ From 0 to all in zero_node_comm_
 8:     (...allocate ring exchange buffers, initialized to basis in a sorted fashion, only
    on ranks 0 of node...)
 9:     // Communication to rank 0 of every node to find size
10:     MPI_Gather(...);                                         ▷ On node_comm_
11:     // IDs for the ring exchange
12:     PetscMPIInt zerosize = -1, zerorank = -1, next = -1, prec = -1;
13:     if zero_node_comm_ != MPI_COMM_NULL then
14:         MPI_Comm_size(zero_node_comm_, &zerosize);
15:         MPI_Comm_rank(zero_node_comm_, &zerorank);
16:         next = (zerorank + 1) % zerosize;
17:         prec = (zerorank + zerosize - 1) % zerosize;
18:     end if
19:     (...broadcast zerosize on node_comm_...)
20:     if node_rank_ then
21:         for PetscMPIInt exc = 0 to zerosize-1 do
22:             MPI_Send(...)                                ▷ Unfound elements
23:             MPI_Recv(...)                               ▷ Computed elements
24:         end for
25:     else
26:         for PetscMPIInt exc = 0 to zerosize-1 do
27:             MPI_Sendrecv_replace_(...);  ▷ Basis exchange using prec and next
28:             // source is required to find global indices
29:             // from nonlocal values
30:             PetscMPIInt source = mod_((prec - exc), zerosize);
31:             (...binary lookup of local elements...)
32:             for i = 1 to node_size_ do
33:                 MPI_Recv(...missing info of rank i)
34:                 (...binary search, if found assign and multiply by -1...)
35:                 MPI_Send(...required info to rank i)
36:             end for
37:         end for
38:     end if
39:     (...flip all the signs: multiply by -1 all elements...)
40: end procedure
```

Chapter 4

Results, performance and discussion

We devote the following sections to expose several results that were obtained studying the time evolution of the physical system under certain conditions and to describe the performance of the application developed.

4.1 Results

4.1.1 Survival probability

We can study the time evolution of the system by means of the survival probability. We start with the system prepared with a given initial state $|\Psi(0)\rangle$ at $t = 0$. The probability of finding the system in state $|\Psi(0)\rangle$ at time t is the so-called survival probability given by

$$F(t) = |A(t)|^2 \equiv |\langle\Psi(0)|\, e^{-iHt}\, |\Psi(0)\rangle|^2 \tag{4.1}$$

$A(t)$ is the survival amplitude. We evaluated this quantity by means of the Krylov subspace methods up to a size of $L = 36$ at half-filling, which has a subspace dimension of over 9 billion. Figure 4.1 exposes a very clean behavior of the survival probability given that we haven't introduced any form of disorder to the system, for these simulations we used the Hamiltonian shown in Eq. (2.1) with $t = 1$ and a weak interaction of $V = 0.2$. The initial state used in these simulations is the Neel ordered state, which in our binary representation is the state given by (...010101). The actual decay behavior of the survival probability can be used as a measure to extract the physical properties of the system.

4.1.2 Disordered systems

We now proceed to evaluate the survival probability of a disordered system by focusing our attention to the Aubry-André model with Hamiltonian operator given

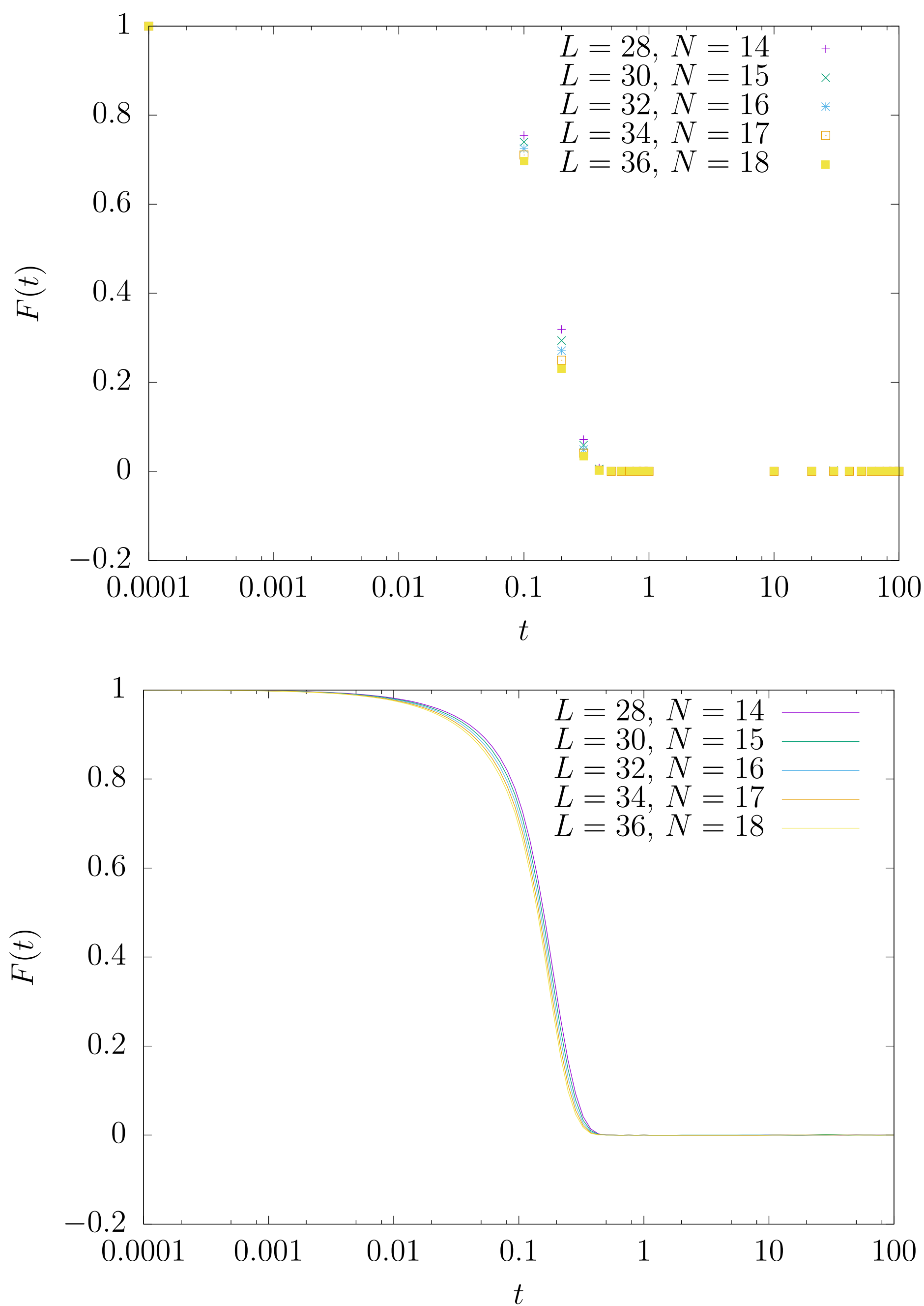

Figure 4.1: Survival probability for different system sizes on a system with no disorder

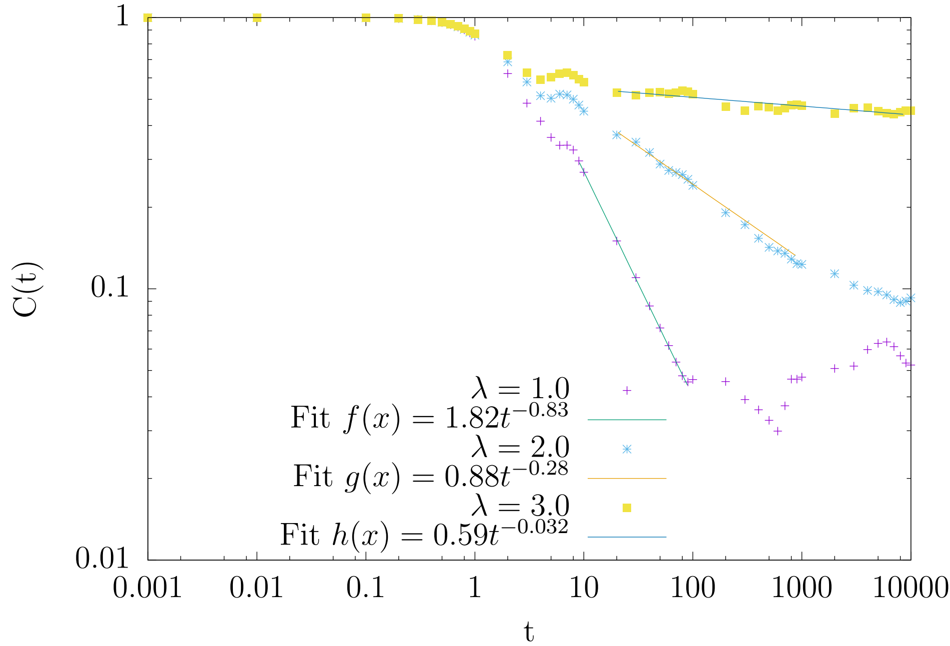

Figure 4.2: Temporal autocorrelation for the Aubry-André model non-interacting case with 1 particle ($L = 55$) using periodic boundary conditions for different values of λ

by

$$H = -t\sum_{i=1}^{L-1}(c_i^\dagger c_{i+1} + H.c) + V\sum_{i=1}^{L-1}(n_i n_{i+1}) + h\sum_{i=1}^{L-1} n_i \cos(2\pi\beta i) \tag{4.2}$$

where we have introduced an on-site potential term. The β factor is the golden ratio given by $\frac{\sqrt{5}-1}{2}$. In this set up we stick to lattices with sizes corresponding to numbers in the Fibonacci sequence so that the oscillating term introduces an aperiodic modulation due to the irrationality of β. This introduces some form of disorder normally known as *quasi-disorder*. We stick to periodic boundary conditions and gather disorder averaging results over a random initial state. We evaluate the so-called *temporal autocorrelation function* instead of the survival probability which is given by

$$C(t) = \frac{1}{t}\int_0^t |\langle\Psi(0)|\,|\Psi(t')\rangle|^2 dt' \tag{4.3}$$

to study the dynamical behavior of the system.
The system is usually studied as a function of the parameters t, V and h in Eq. (4.2). We can define $\lambda \equiv h/t$ and change this parameter for a fixed value of V.

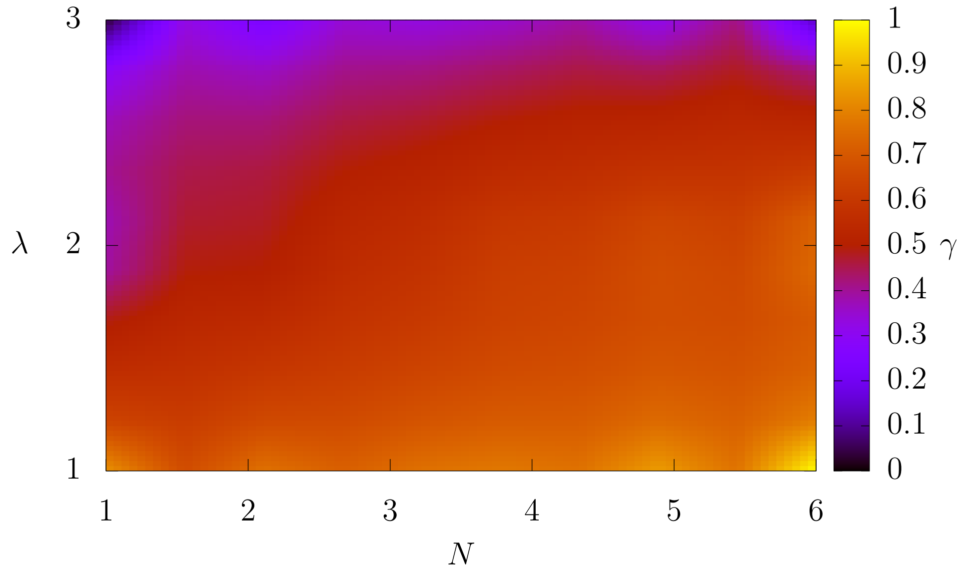

Figure 4.3: Temporal decay exponent γ for different number of particles (N) and values of λ for the interacting Aubry-André model with periodic boundary conditions and $L = 55$

Figure 4.2 shows the behavior of the temporal autocorrelation function for three different regimes in the non-interacting case for a single particle: $\lambda < 2$ exposes the behavior of the delocalized states or extended states, $\lambda > 2$ shows the localized state behavior and $\lambda = 2$ is known to be a critical state, neither localized nor extended. [6]
For $\lambda = 2$, the system is know to be in a critical phase with a very rich energy spectra with fractal properties. The temporal decay of the system in this phase will highly depend on the initial state of the system. For the extended states and the localized states our results are in full agreement with those shown by [6], however for the critical phase a temporal decay exponent (γ) of -0.14 has been reported with a localized initial wave packet; while we average over several random initial states and obtained an average value of -0.27 for the decay exponent. We believe the difference is due to rich spectrum inherent to the critical phase of the system and it's dependency on the initial state chosen.

An interesting case of study is to evaluate the temporal evolution of the system described with the Aubry-André model for the interacting case, using higher densities. We proceed in a similar fashion as we did for the single particle case and study the dynamics of the system using increasing densities. For this section we provide a perspective on the results obtained, and each of the simulations done are presented in the Appendix section of this document.

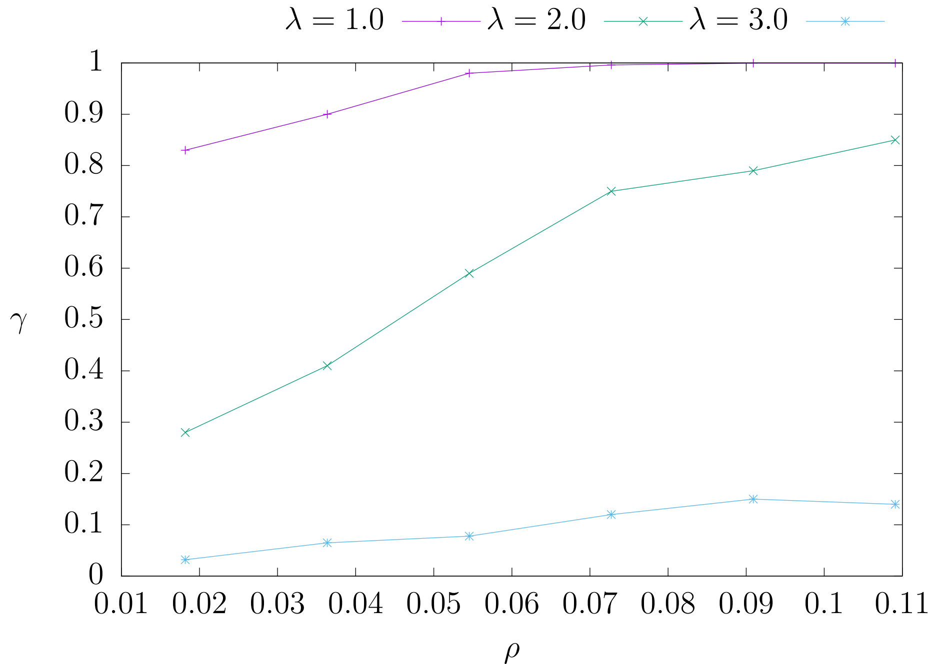

Figure 4.4: Temporal decay exponent γ for different densities ($\rho \equiv N/L$) and values of λ for the interacting Aubry-André model with periodic boundary conditions and $L = 55$

Figures 4.3 and 4.4 shows a perspective of the temporal decay exponent as a function of both λ and the density, it can be seen from this and from the results shown in the Appendix that increasing the density of system favors the decay towards the extended states. In the language of Ref. [13], it can be said that increasing the density favors thermalization; although according to [13] the regime for which $0 < \gamma < 1$ no thermalization occurs at all.
It can also be extracted from our analysis that $\gamma \leq 1$ for the λ values considered. This is an indicator that the powerlaw exponent extracted behaves and scales with similarity with the participation ratio PR

$$PR \propto \mathcal{D}^{\gamma} \tag{4.4}$$

where $\mathcal{D}$ is the dimension of the Hilbert space. We can interpret the illustrated regimes of the system as follows:

- $\gamma \to 1$: As illustrated for increasing densities and $\lambda = 1.0$, indicates the presence of chaotic initial states and diffusive dynamics
- $\gamma < 1$: For $\lambda = 2.0$ and $\lambda = 3.0$, indicates the presence of non-chaotic delocalized states and subdiffusive dynamics

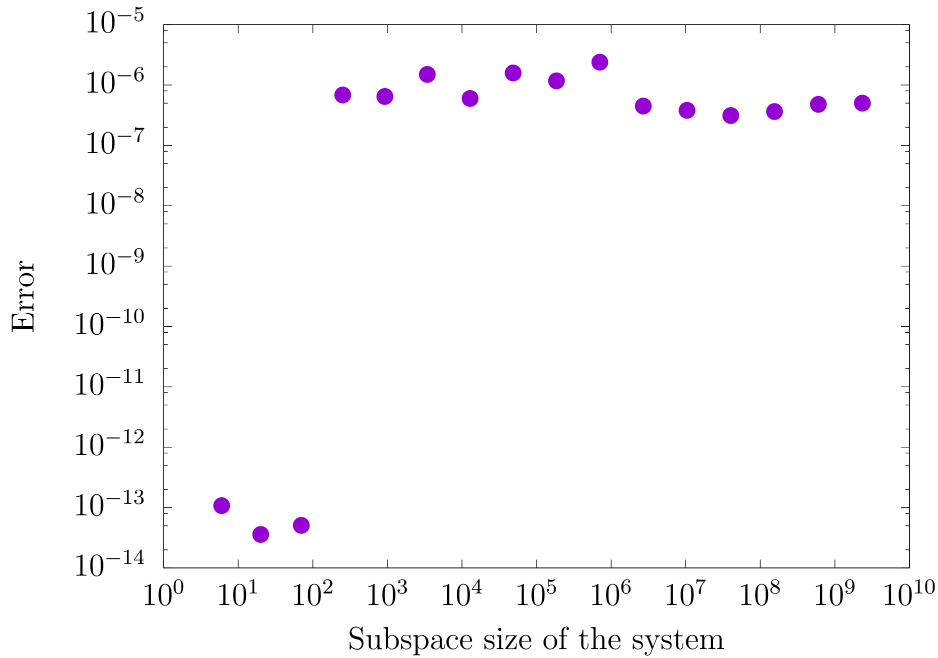

Figure 4.5: Representation of the absolute error for the time evolution of an unperturbed clean system as a function of the subspace dimension of the quantum system

The system can be studied further in light of the results presented in Figure 4.3 to, for example, extract the regime for which the system enters a critical state.

4.1.3 A perspective on the absolute error

Given that our approach relies on projections on the Krylov subspace with a given dimension on it's own, a form of *truncation* error is expected as we keep this dimension to a finite value. We can evaluate this absolute error by using an initial state and evolving the system up to a certain time value t, the resulting state is then evolve to time $-t$. For an unperturbed clean system (such as the ones used for time evolution in Figure 4.1), we should obtain the initial state with a given tolerance set internally as a numerical parameter in the mathematical method.

Figure 4.5 puts this into perspective for increasing system sizes. For our purposes we set the internal numerical tolerance of the Krylov subspace method to 10^{-7} and the dimension of the Krylov subspace to 30, this value though, can be increased or reduced depending on the nature of the calculations. In Figure 4.5 we plot the norm of the difference between the initial state and the final state after evolving to time t and returning by using time $-t$

$$error = || \left|\Psi(0)\right\rangle - e^{iHt}e^{-iHt} \left|\Psi(0)\right\rangle ||_2 \tag{4.5}$$

The results show what we expect from the Krylov subspace approach and the descriptions discussed in [9] and even for very large subspace sizes of the system the global error in the time evolution stays within tolerance.

4.2 Performance

4.2.1 Systems

Both the serial and parallel application (including all it's versions) have been enabled and tested in three different computational environments. Table II shows the description of each of the systems in which the application was tested and used for production runs.

Table II. Systems.

Description	Ulysses-SISSA	Galileo-CINECA	Marconi-CINECA
Model	—	IBM NeXtScale	Lenovo NeXtScale
Architecture	x86_64	Linux IB Cluster	Intel OmniPath Cluster
Nodes	—	516	1 512
Processors	10-cores Intel Xeon E5-2680 v2 2.80 GHz (2 per node)	8-cores Intel Haswell 2.40 GHz (2 per node)	18-cores Intel Xeon E5-2697 v4 2.30 GHz (2 per node)
RAM	40 GB/node	128 GB/node	128 GB/node
Internal Network	—	IB with 4x QDR switches	Intel OmniPath

4.2.2 Compilers and libraries

The application has been compiled and tested using the C++ MPI wrappers of OpenMPI version 1.8.3 above GNU GCC 4.9.2 and the Intel MPI compiler from the 2016 package version 5.1.3. We expose the results obtained with the latter. Different versions of Intel MKL and OpenBLAS have been tested also for underlying linear algebra operations, we show the results obtained with Intel MKL version 11.3.3. The application was tested with OpenBLAS at the beginning, since this is the default linear algebra library installed with PETSc if another is not specified. The versions of the higher-end libraries used are:

- Boost 1.61.0
- PETSc 3.7.3
- SLEPc 3.7.2

Configuration has been discussed in Section 3.2.

4.2.3 Performance results

Our objective in this section is to present the performance of the application using a single environment to evaluate the application itself, so, unless otherwise stated

we present the results using the *Galileo-CINECA* machine shown in Table II. We decided to use Galileo as our main testing environment, given that Ulysses-SISSA provides less resources in terms of memory than the ones required to undertake some of the simulations presented here. Furthermore, although Marconi has more updated hardware and a better network (OmniPath), this project was developed during the early stages of the machine, shortly after it was enabled. Due to this, some results (particularly the ones resulting from simulations with a large amount of nodes) were not reproducible. We have chosen Galileo for our performance results, given that during development of the project this machine provided the most stable results in terms of performance.
We proceed to compare each of the versions described in the previous chapter using the same configuration overall, with the Hamiltonian shown in Eq. (2.1).

Replicated basis version

We start by evaluating the performance of the first instance of the application in a parallel environment. As was described in the previous chapter this version uses basis replication among all processing elements to construct the matrix representation of the Hamiltonian matrix, however, the computation and distribution of this object and the later time evolution step are performed in a fully distributed fashion. Figure 4.6 shows the strong scaling behavior of the application in the two most important steps: the construction of the Hamiltonian object and the time evolution.

It can be seen that the time of the construction of the Hamiltonian object is very small compared to the actual time evolution procedure even for a relatively small physical time parameter ($t = 100$). This constituted our base for following development: one of our objectives is to keep the time to construct the Hamiltonian object small when compared to the time evolution, in order to avoid compromising scalability for a very large number of processing elements. Our results show a good scaling behavior even for large amount of processing elements (2048 MPI processes mapped to each core, as shown in the last bar in Figure 4.6). An MPI approach was also required, given that for larger problem sizes ($L = 30$ at half filling and larger, for instance) can't be solved by means of a single computational node because of memory requirements and to be able to use more than one computational node for calculations.

Another important fact that can be extracted is that constructing and solving the system for the given parameters takes under 2 minutes overall with 128 nodes on the Galileo-CINECA machine. This is an important factor if for instance, one is interested to undertake disorder averages over random initial states or another random parameter of study.
A big improvement was obtained when using Intel MKL for underlying linear algebra operations instead of OpenBLAS, this was done by properly linking PETSc/SLEPc with the library. In Figure 4.6 we present the results obtained with MKL. Given

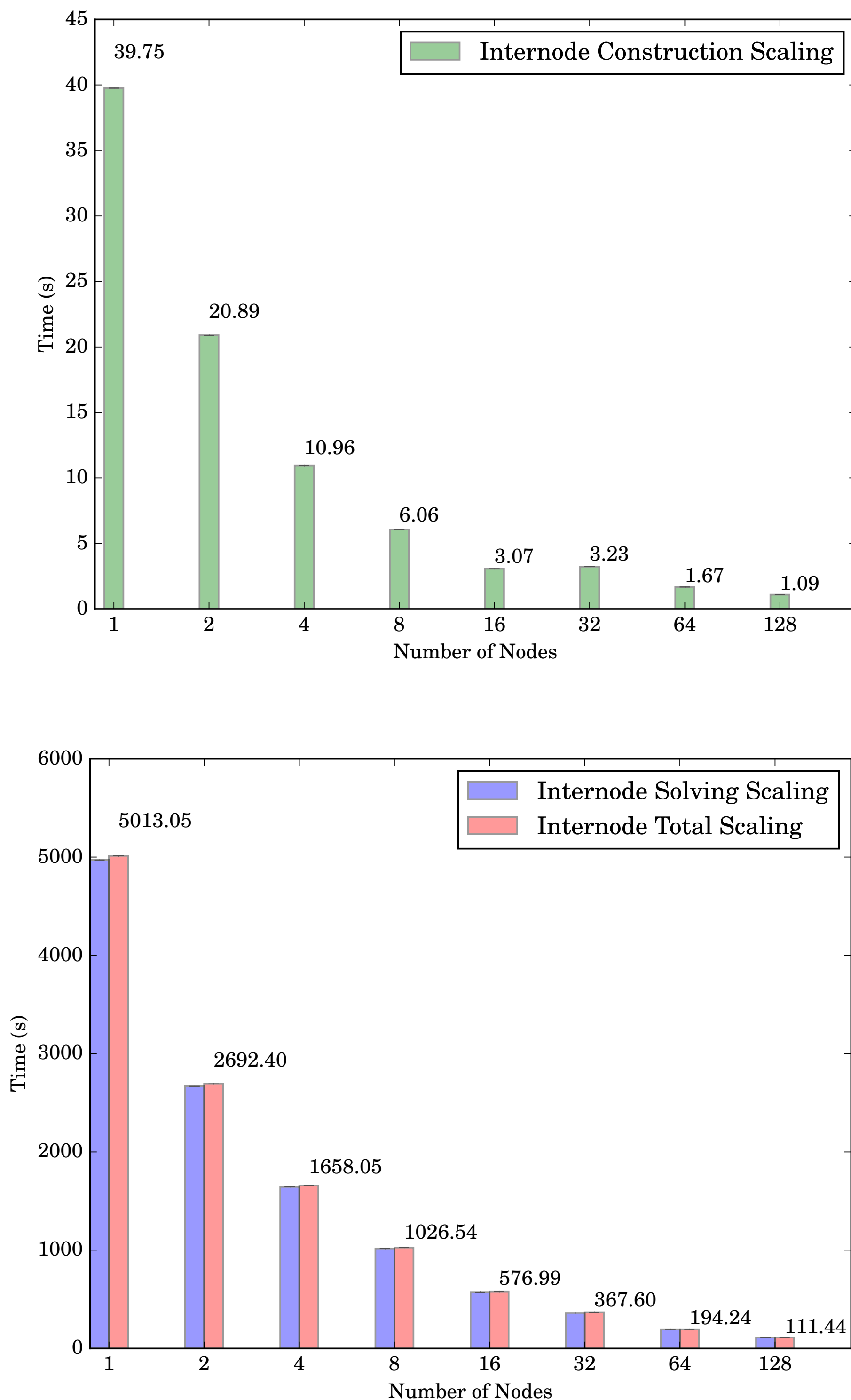

Figure 4.6: Running time of the application for the construction of Hamiltonian, time evolution and overall time using a system with $L = 28$ at half-filling (subspace dimension of 40 116 600) up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (replicated basis version)

that the method used for time evolution relies heavily on linear algebra operations, we conclude that using MKL is highly beneficial as the library is highly optimized for the architecture used.

Node communicator version

As was described in the previous section, basis replication is a big issue when it comes to large problem sizes. We developed the `Node communicator` version with two objectives in mind: avoid basis replication and therefore have the possibility of studying larger system sizes and retain strong scaling behavior in the process. With this development we managed to achieve both.
On the Galileo-CINECA machine we managed to solve a system with size up to $L = 34$ at half filling using this approach, although this version is also very useful on machines with less amount memory per node given that full replication of the basis is avoided and actual computation and scalability of the overall application is not compromised. The actual time evolution of the system is unchanged from the previous version so Figure 4.7 shows the same behavior as the previous version. In Figure 4.7 we can also see that the increase in time required to construct the Hamiltonian operator is very small compared to the time evolution of the system even for this relatively big system size, which was exactly our goal. Other different approaches, such as a systematic re-computation of sections of the basis to avoid storing this object in memory, proved to be inefficient.
Strong scaling behavior for this approach is not linear, but the results obtained are satisfactory when it comes to actually solving the problem for a large amount of processing elements. A *flat* behavior can be seen at the transition from 16 to 32 nodes, we believe this could be related to the network arrangement of the machine so that communication beyond 16 nodes could be using different switches in the interface (higher tree in the network arrangement); though this would have to be investigated further in order to ascertain.

Ring exchange version

The ring exchange version was designed with the purpose to open the possibility to even larger problem sizes. Recalling the description in the previous chapter, in this version no memory replication occurs in the application whatsoever. Memory requirements decrease linearly with increasing number of computing nodes, but *time scalability* is indeed compromised given that a larger amount of processing elements implicates more overall communications. So in perspective, a strong scaling view of the performance like the one shown for the previous developments is not a meaningful for this approach.
With this version, our goal was to have the ability to study even larger systems sizes by means of a full distribution of memory and computation overall. To demonstrate a perspective of the performance accomplished by this approach we show the time of computation required in the most important sections of the application with in-

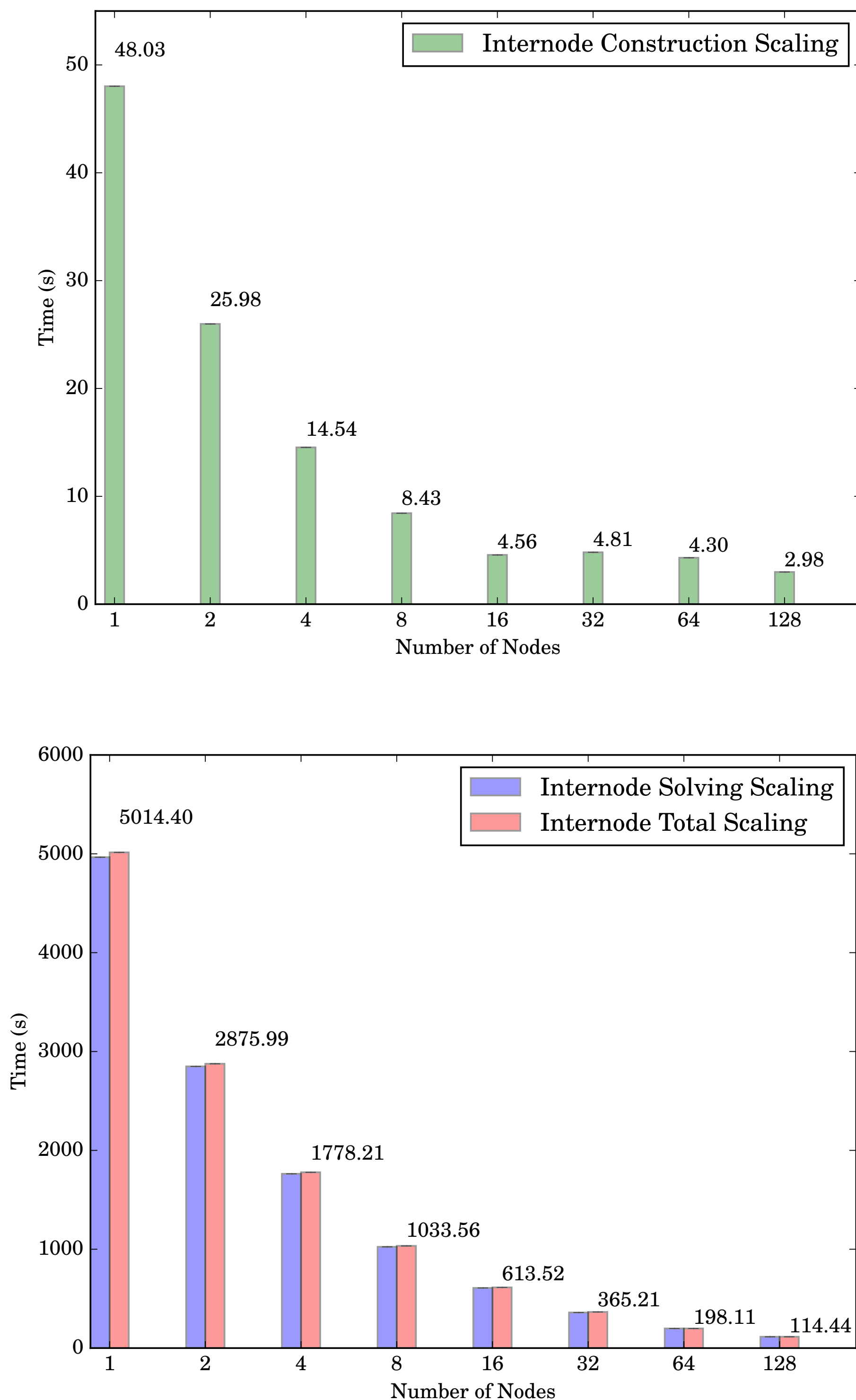

Figure 4.7: Running time of the application for the construction of Hamiltonian, time evolution and overall time using a system with $L = 28$ at half-filling (subspace dimension of 40 116 600) up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (`Node comm` version)

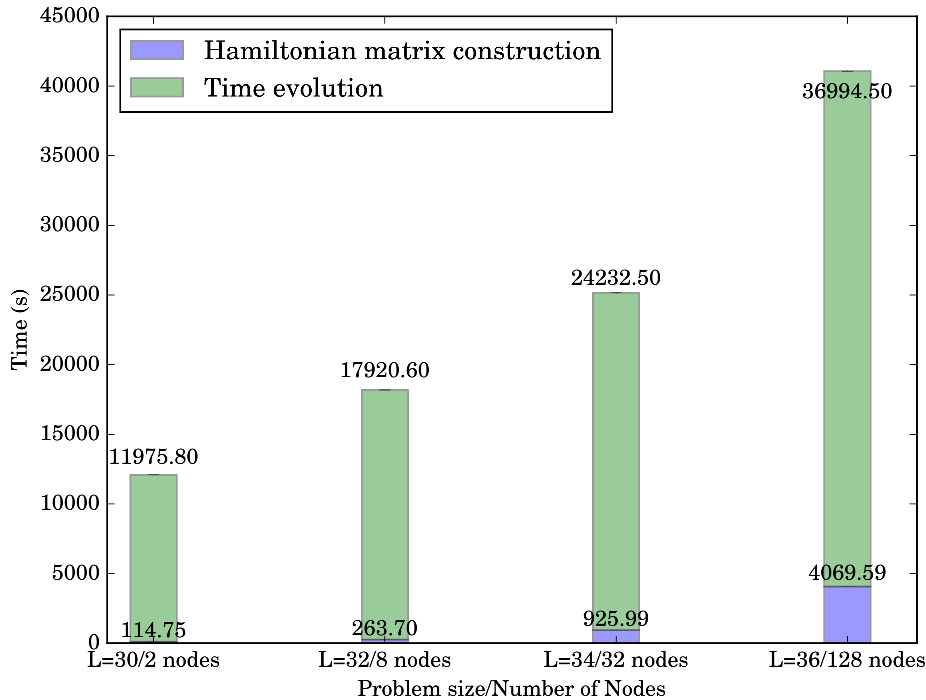

Figure 4.8: Running time of the application for the construction of Hamiltonian and time evolution using a different problem sizes at half-filling up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (`Ring exchange` version)

creasing problem size and processing elements in Figure 4.8.

It can be seen that the time to construct the Hamiltonian matrix object becomes important as the problem size increases, up to roughly 10% of the overall computation time for the larger case ($L = 36$ at half-filling). With this approach, most of the communication done during the construction of the Hamiltonian object occurs inside of the node which is beneficial in general terms, but when using a large number of processing elements the communication steps start to a considerable percentage of the overall execution time.
Even so, with this approach we were able to solve the $L = 36$ at half-filling system; a system that roughly has a subspace dimension of more than 9 billion elements. Memory requirements to solve this system are very large as was estimated and presented in Table I. Using this approach we were able to calculate the survival probability with a tolerance of 10^{-7} shown as shown in Figure 4.1.

Combined Ring-Node communicator version

The previous `Ring exchange` version is presented as a good solution towards obtaining a solution to the problem for very large systems sizes. This is our *go-to* approach for systems that have large subspace dimension. Even for a big number of processing elements (2048 MPI processes in our example), the time to construct the Hamiltonian object accounts for 10% of execution time of the application for the larger example tested.
This version was developed as an attempt to reduce the execution time of the construction of the Hamiltonian section of the application by means of another distribution of the basis and a different communication pattern. In comparison to our previous `Ring exchange` version, this approach uses more memory resources in storing the elements of the basis, as some replication occurs in the new distribution. Figure 4.9 shows a similar representation to the one shown for the `Ring exchange` version for the sake of comparison. It can be seen that as the number of processing elements and size of the problem, instead of a speedup we have achieved an underperforming behavior when compared to the previous version. The reason behind it has been mentioned previously: in order to avoid exhausting memory resources for a given node, the same memory segment has to be used for the intranode communication between each of the MPI processes of the node and the first MPI process of the same node. This has to be performed in such a way because of the large amount of elements in the communication, in turn, a sequential communication pattern is done.
This effect compromises performance in relation to the previous `Ring exchange` version, in which each MPI process has a more balanced workload. We conclude that the `Ring exchange` version provides a better approach when it comes to solving very large problem sizes.

4.2.4 Strong scaling

Here we present the speedup of the application for the two versions that were designed to scale in time. As previously stated, for the Ring-Exchange version and the Combined Ring-Node communicator version, Figures 4.8 and 4.9 present a better perspective on the performance of the application.

As expected, the version with the replicated basis provides the best scaling in time with the number of computational nodes, even though the system presented here to evaluate performance is relatively small compared to the biggest system we were able to simulate, the application continues to scale even up to 128 nodes for both the construction of the Hamiltonian and the time evolution steps.

On the `Node comm` version, the time scaling for the time evolution step remains the same. However, in Figure 4.11 we see that the construction of the Hamiltonian matrix stops scaling after 16 nodes. This is related to the size of the system: with

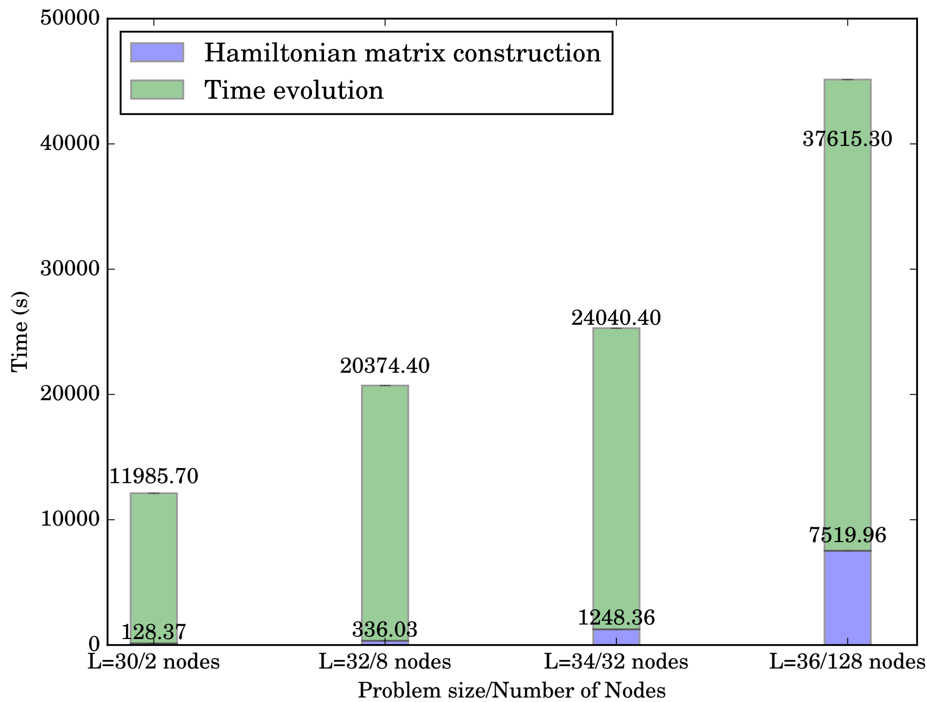

Figure 4.9: Running time of the application for the construction of Hamiltonian and time evolution using a different problem sizes at half-filling up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (`Combined Ring-Node comm` version)

smaller system sizes one reaches the so-called Amdahl's law quicker as the number of computing nodes increases. In order to provide evidence of this, Figure 4.12 shows the strong scaling of the construction of the Hamiltonian section of the application for the $L = 30$ at half-filling system.

Since the system with $L = 30$ at half-filling is too large memory-wise to be simulated on a single computational node, the strong scaling is measured based on the time to construct the Hamiltonian representation on 2 computing nodes in Figure 4.12. It can be seen that for a larger system the communication algorithm implemented using MPI continues to scale even up to 128 nodes.

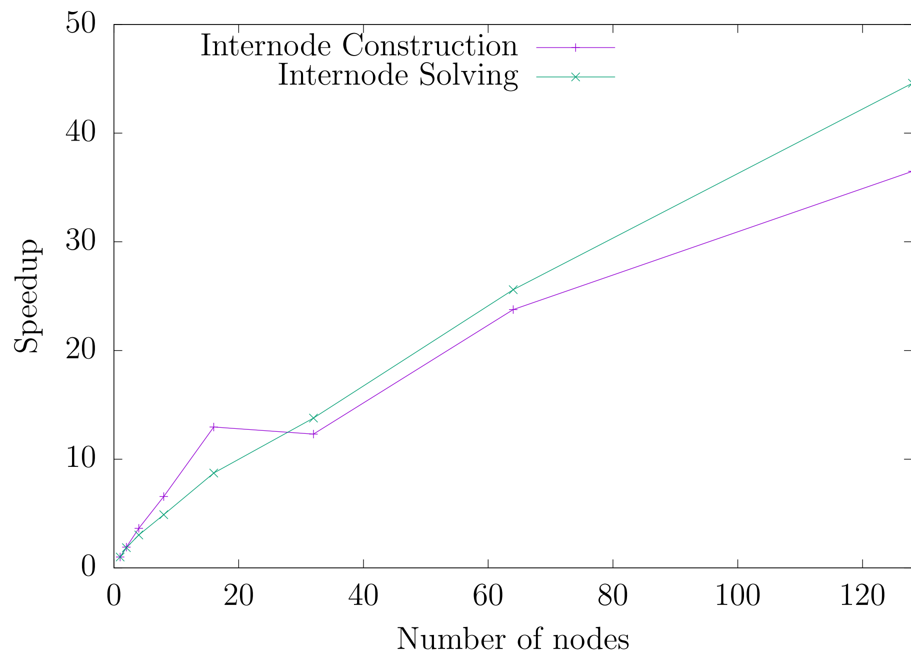

Figure 4.10: Speedup of the application for the construction of Hamiltonian and time evolution using a system with $L = 28$ at half-filling (subspace dimension of 40 116 600) up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (replicated basis version)

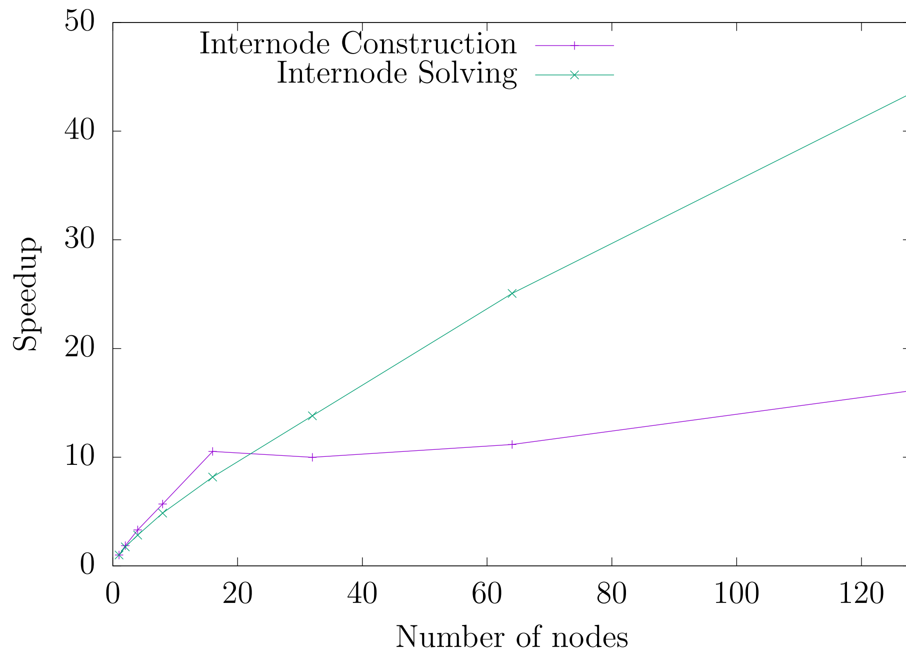

Figure 4.11: Speedup of the application for the construction of Hamiltonian and time evolution using a system with $L = 28$ at half-filling (subspace dimension of 40 116 600) up to $t = 100$ with a tolerance of 10^{-7} for a different amount of computing nodes (`Node comm` version)

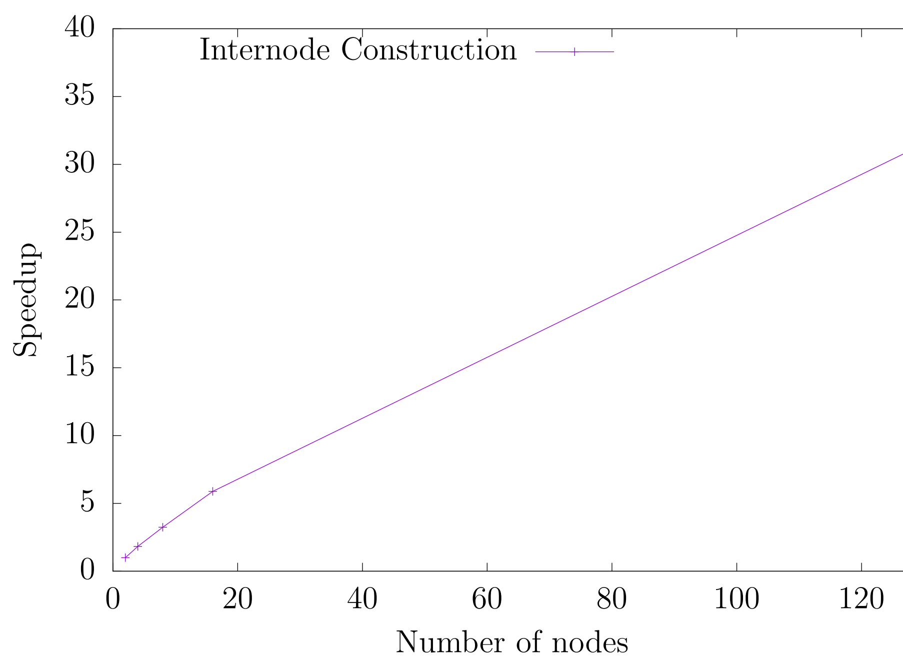

Figure 4.12: Speedup of the construction of the Hamiltonian step using a system with $L = 30$ at half-filling (subspace dimension of 155 117 520) for a different amount of computing nodes, referenced to two computing nodes (`Node comm` version)

Chapter 5

Future Work

The memory problem that is entailed by basis replication among processing elements in a parallel environment has been addressed in this work by means of distribution and MPI communication patterns. This has allowed us to study the dynamics of systems with a very large subspace dimension. A caveat of the method employed is the fact that the Hamiltonian matrix needs to be stored in memory to perform the time evolution and this would have to be addressed if one is interested in studying even larger systems, as even with a full distribution the memory requirement is too large. A possible solution to be evaluated would be an *on-the-fly* construction of the Hamiltonian matrix and intermediate operations of the time evolution, although this procedure would have to be highly optimized to obtain results in a timely manner. This would also imply a specialization of the routines to specific Hamiltonian operators, which means that if one is interested in studying systems with a different Hamiltonian a much bigger effort would have to be committed and versatility would be reduced.

The results presented here were limited by the amount of computing resources that a user can allocate on the systems described: 128 nodes on Galileo and 166 nodes on Marconi. However, a large scale simulation using larger computing resources can be undertaken if enough scientific motivation is proposed.

Heterogeneous computing using GPGPUs or co-processors can provide a viable framework to study the dynamics of the system. With pending evaluation, this solution might be able to provide a good performing method to study not large systems (due to memory consumption reasons) but perhaps smaller systems for very large values of time. PETSc already provides the background to perform this task, but the complex datatypes that are required to solve the problem might provide difficulties with some of the routines (namely, SLEPc's BV class). Another solution if one is interested in using GPGPUs would be to re-implement the application exposed in the Section 3.3 (Serial version) using specialized libraries such as cuSPARSE of CUSP.

Chapter 6

Conclusion

An application to study the dynamics of quantum correlated systems suitable to be executed on massively parallel supercomputers has been developed and tested in this work. We have used high-performing libraries in conjunction with MPI distribution algorithms in order to study large quantum systems with subspace dimension of over 9 billion states with the computational resources available.
To check the validity of the results we have studied and presented the dynamics of known models. [13][6]
We have studied the performance of the application with different MPI distribution schemes using equal physical systems and concluded that the best approach for moderate system sizes is the `Replicated basis` or `Node Communicator` subject to available computational resources; while the `Ring Exchange` method proposes an efficient and simple approach to study the dynamics of very large problem sizes.
The work presented here constitutes an instance for which an HPC approach is most useful towards scientific computing and the development of cutting-edge scientific results.

References

[1] Argonne National Laboratory. (2016) *PETSc Users Manual.* Revision 3.7. Mathematics and Computer Science Division, UChicago, Illinois.

[2] Boost C++ Libraries. Aug. 2016 `http://www.boost.org`

[3] Eisert, J. Friesdorf, M. Gogolin, C. (2015) *Quantum many-body systems out of equilibrium* Nature Physics 11, Pp: 124-130.

[4] Fehske, H. Schneider, R. Weisse, A. (2008) *Computational Many-Particle Physics.* Lect. Notes Phys. 739. Springer, Berlin Heidelberg, Pp: 529-544.

[5] Iyer, S. Refael, G. Oganesyan, V. Huse, D. (2013) *Many-Body Localization in a Quasiperiodic System.* ArXiv preprint arXiv:1212.4159v2

[6] Ketzmerick, R. Petschel, G. Geisel, T. (1992) *Slow decay of temporal correlations in quantum systems with Cantor spectra.* Physical Review Letters, Issue 5, Volume 69.

[7] Moler, C. Van Loan, C. (2003) *Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later* SIAM Review, 45-1, Pp: 3-000.

[8] Prelovsek, P. Barisic, O.S. Znidaric, M. (2016) *Absence of full many-body localization in disordered Hubbard chain.* ArXiv preprint arXiv:1610.02267v1.

[9] Sidje, R. (1998) *Expokit: A software package for computing matrix exponentials.* ACM Trans. Math. Softw., 24-1, Pp: 130-156.

[10] Siro, T. Harju, A. (2015) *Exact diagonalization of quantum lattice models on coprocessors.* Computer Physics Comunications, arXiv 1511.00863v1.

[11] Roman, J. Campos, C. Romero, E. Tomás, A. (2016) *SLEPc Users Manual.* DSIC-II/24/02. Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia, España.

[12] Távora, M. Torres-Herrera, E.J. Santos, L. (2016) *Inevitable Powerlaw Behavior of Isolated Many-Body Quantum Systems and How It Anticipates Thermalization.* ArXiv preprint arXiv:1601.05807v3.

[13] Távora, M. Torres-Herrera, E.J. Santos, L. (2016) *Powerlaw Decay Exponents: a Dynamical Criterion for Predicting Thermalization*. ArXiv preprint arXiv:1610.04240v1.

[14] Troyer, M. (2006) *Computational Physics*. ETH Zurich, Lecture Notes.

[15] Varma, V.K. Lerose, A. Pietracaprina, F. Goold, J. Scardicchio, A. (2016) *Energy diffusion in the ergodic phase of a many body localizable spin chain*. ArXiv preprint arXiv:1511.09144

Appendix A

Interacting Aubry-André model results

We devote this appendix section to show the results obtained for each of the simulations related to the interacting case of the Aubry-André model for different densities.

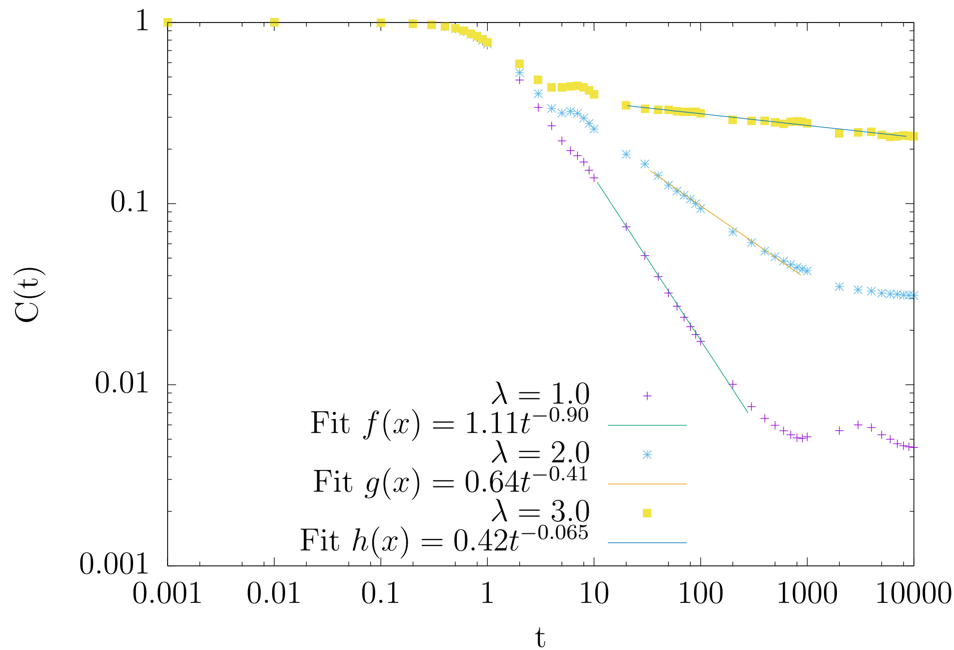

Figure A.1: Temporal autocorrelation for the Aubry-André model interacting case with 2 particles ($L = 55$, $\mathcal{D} = 1485$) using periodic boundary conditions for different values of λ (200 realisations)

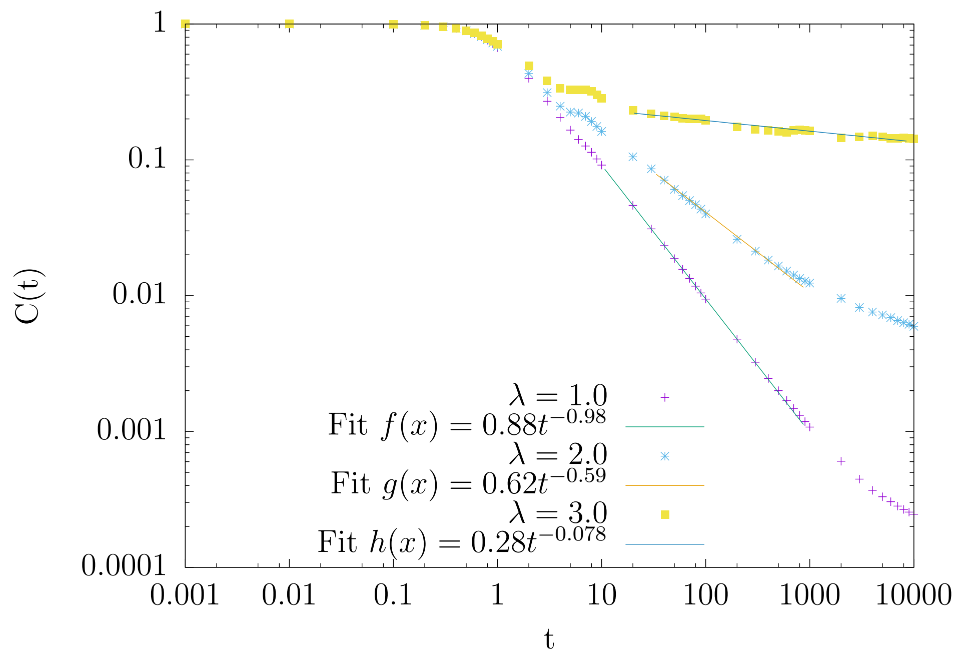

Figure A.2: Temporal autocorrelation for the Aubry-André model interacting case with 3 particles ($L = 55$, $\mathcal{D} = 26235$) using periodic boundary conditions for different values of λ (200 realisations)

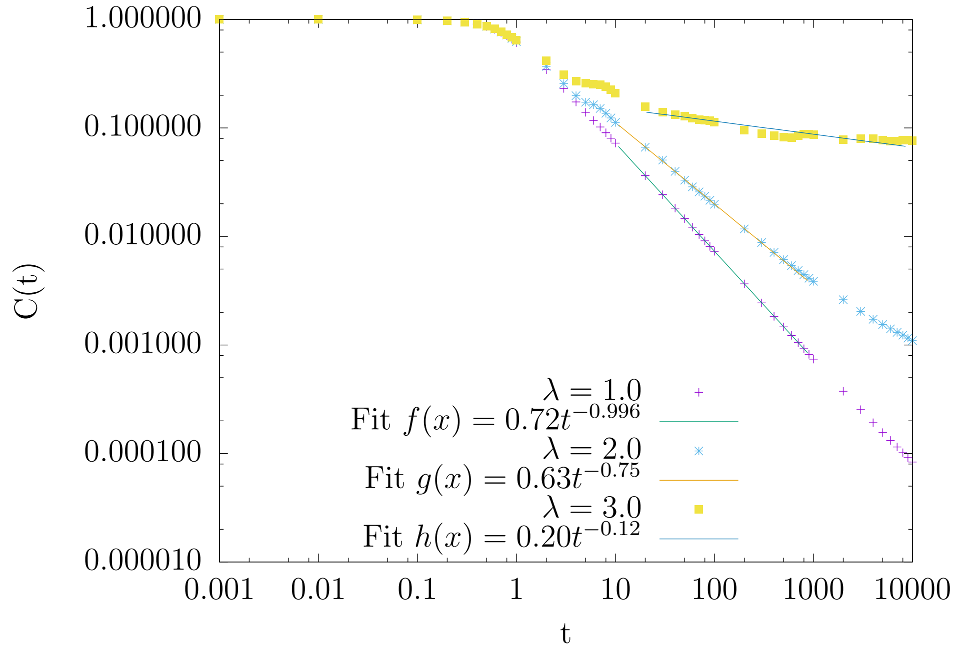

Figure A.3: Temporal autocorrelation for the Aubry-André model interacting case with 4 particles ($L = 55$, $\mathcal{D} = 341055$) using periodic boundary conditions for different values of λ (100 realisations)

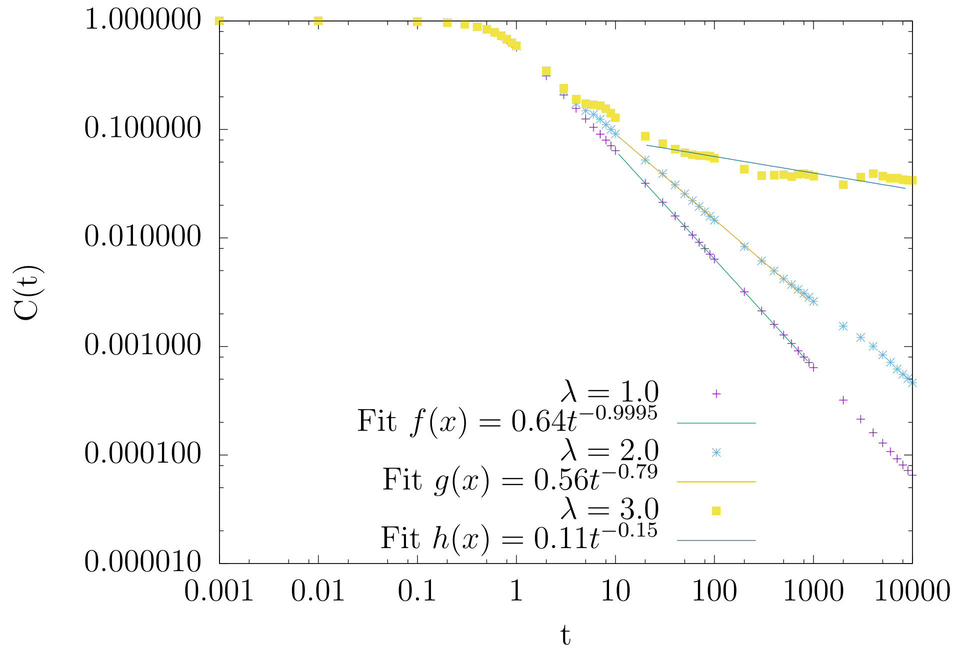

Figure A.4: Temporal autocorrelation for the Aubry-André model interacting case with 5 particles ($L = 55$, $\mathcal{D} = 3478761$) using periodic boundary conditions for different values of λ (25 realisations)

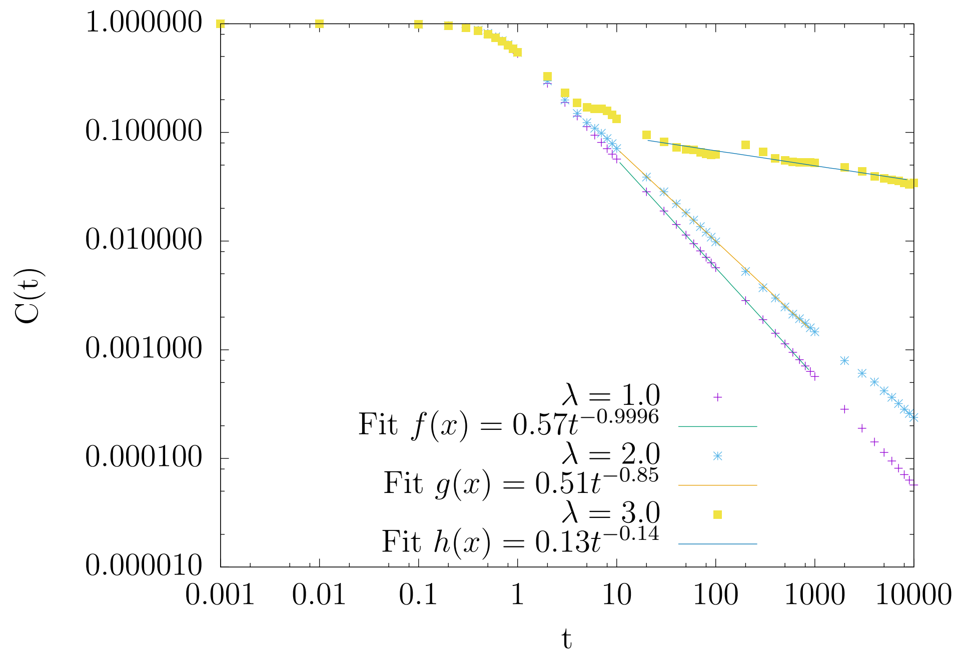

Figure A.5: Temporal autocorrelation for the Aubry-André model interacting case with 6 particles ($L = 55$, $\mathcal{D} = 28989675$) using periodic boundary conditions for different values of λ (5 realisations)