BERGISCHE
UNIVERSITÄT
WUPPERTAL

Matrix QM: Lanczos Algorithm Implementation for 1D Quantum Mechanics

Hybrid MPI + OpenMP parallelization with domain decomposition

Course:

Numerical Methods in Classical Field Theory and
Quantum Mechanics (NM2b)

Group Members:

Dhruv Patel (2130292)
Mohammadreza Khansari (2132180)

Advisor:

Dr. Tomasz Korzec

Fakultät für Mathematik und Naturwissenschaften
Bergische Universität Wuppertal

April 9, 2025

Abstract

This term paper presents a numerical framework for solving eigenvalue problems in 1D quantum mechanics using a matrix formulation and the Lanczos algorithm. Focusing on the harmonic oscillator potential, we develop serial and parallel (MPI + OpenMP) versions of the algorithm that efficiently handle large, sparse Hamiltonian matrices without explicit matrix construction. The discretization scheme employs finite difference approximations for the momentum operator, enabling $O(N)$ complexity matrix-vector operations. Validation against analytical solutions of the quantum harmonic oscillator demonstrates exact agreement for low-lying energy states of the large Hermitian matrices. Parallelization strategies using domain decomposition and nearest-neighbour communication are analyzed through strong scaling tests, showing near-linear speed-up for large system sizes. The methodology provides a robust foundation for studying complex, high-dimensional quantum systems while maintaining computational efficiency through sparse matrix techniques and distributed computing.

Contents

Chapter 1

Introduction

Quantum mechanics fundamentally describes microscopic systems through the Schrödinger equation, a partial differential equation that governs the evolution of the wave function. Although analytical solutions exist for simple potentials, such as the harmonic oscillator, most realistic systems require numerical treatment. In this work, we discretize continuous quantum operators—like momentum and position—into finite-dimensional matrices via second-order finite differences, transforming the Schrödinger equation into a matrix eigenvalue problem that is computationally tractable.

The matrix formulation becomes essential when dealing with complex potentials or scalable simulations of higher-dimensional systems, where traditional basis-set methods have limitations. However, the resulting Hamiltonian matrices surge with system size, necessitating efficient algorithms like the Lanczos method that exploit sparsity patterns and their tridiagonal structure for $O(N)$-complexity matrix-vector operations. Our implementation optimizes the quantum harmonic oscillator as a benchmark system, leveraging its known analytical solutions for validation.

For performance advancement, we extend our numerical framework to parallel computing using MPI and OpenMP. Our strategy tackles two main challenges: (1) efficiently representing sparse operators on distributed memory architectures and (2) ensuring algorithmic stability through full re-orthogonalization in parallel environments. This approach scales effectively on modern high-performance computing systems while maintaining the physical accuracy required for studying complex quantum systems.

Chapter 2

Theory

2.1 From Classical to Quantum Mechanics

In classical mechanics, the dynamics of a one-dimensional system are described as:

$$\frac{dx}{dt} = \frac{\partial H}{\partial p}, \quad \frac{dp}{dt} = -\frac{\partial H}{\partial x}, \tag{2.1}$$

with the Hamiltonian given by

$$H(p, x) = \frac{p^2}{2m} + V(x, t). \tag{2.2}$$

Although classical mechanics effectively describes macroscopic phenomena, it is inadequate in several key areas where quantum effects emerge:

- **Atomic and Subatomic Scales:** At quantum scales, the classical picture of well-defined particle trajectories breaks down. For instance, classical mechanics inaccurately predicts stable electron orbits around nuclei, whereas quantum mechanics describes electrons as probabilistic wave functions.
- **Wave-Particle Duality:** Classical theory cannot reconcile the dual nature of particles (e.g., electrons, photons) exhibiting both wave-like (interference) and particle-like (localized collisions) properties.
- **Photoelectric Effect:** In classical wave theory, electron emission tends to depend solely on light intensity. However, experiments reveal that emission occurs only when the incident light exceeds a threshold frequency, a phenomenon accurately explained by associating photon energy with $E = h\nu$.
- **Relativistic Regimes:** At velocities approaching the speed of light or in strong gravitational fields, classical mechanics fails, necessitating Einstein's theories of relativity.

Quantum mechanics replaces the deterministic framework of classical mechanics with a probabilistic description [2, 8]. The state of a quantum system is represented by a wave function $\psi(x, t)$, and physical observables are described by operators. It replaces canonical variables with operators satisfying the commutation relation $[\hat{x}, \hat{p}] = i\hbar$, leading to the evolution of the system by time-dependent Schrödinger equation as:

$$i\hbar\frac{\partial}{\partial t}\psi(x, t) = \hat{H}(\hat{p}, \hat{x})\psi(x, t), \tag{2.3}$$

where the Hamiltonian operator $\hat{H}$ typically comprises a kinetic energy term and a potential energy term. For a one-dimensional system, the Hamiltonian operator is

$$\hat{H}(\hat{p},\hat{x}) = \underbrace{\frac{\hat{p}^2}{2m}}_{\text{kin. energy}} + \underbrace{V(\hat{x},t)}_{\text{pot. energy}}, \tag{2.4}$$

with the momentum operator defined by

$$p \to \hat{p} = -i\hbar\frac{d}{dx}. \tag{2.5}$$

2.2 Stationary States and Eigenvalue Problem

When the potential $V(x)$ is time-independent, one can employ the separation ansatz $\psi(x,t) = \phi(x)e^{-iEt/\hbar}$, which transforms the time-dependent Schrödinger equation into the stationary Schrödinger equation:

$$\hat{H}\phi_n(x) = E_n\phi_n(x). \tag{2.6}$$

Here, eigenvalues E_n denote the quantized energy states, and $\phi_n(x)$ are the corresponding eigenfunctions. For example, in the harmonic oscillator where $V(x) = \frac{1}{2}m\omega^2x^2$, the eigenvalues and eigenfunctions are given by

$$E_n = \hbar\omega\left(n+\frac{1}{2}\right), \quad \phi_n(x) \propto H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right)e^{-m\omega x^2/2\hbar}, \tag{2.7}$$

where H_n represents the Hermite polynomials.

2.3 Matrix Formulation and Discretization

For solving the Schrödinger equation numerically, the discretization is in a continuous spatial domain over a finite interval—often with periodic boundary conditions—thereby converting differential operators into matrices. Two common approaches for discretizing the second derivative (which is associated with $\hat{p}^2$) include:

- **Finite Difference Method:** This approach approximates the second derivative using nearest-neighbour approximation and yields a sparse, tridiagonal matrix, with error terms proportional to a^2, where a is the lattice spacing.
- **Spectral (Fourier) Method:** Constructs a dense matrix based on the spectral derivative. This method leverages the sampling theorem to reproduce continuum eigenvalues with negligible discretization error accurately [9].

In either case, the discretized Hamiltonian is represented as a Hermitian matrix whose eigenvalues correspond to the energy levels of the system. Discretizing space into N points, where $x_k = ka$ for $k = -M,\ldots,M$ (with periodic boundary conditions), transforms operators into matrices. For example:

- **Position Operator:** Represented by the diagonal matrix $X_{kl} = x_k\delta_{kl}$

- **Momentum Operator:** Finite-difference approximation

$$P_{kl}^2 = \frac{\hbar^2}{a^2} \begin{cases} 2 & k = l \\ -1 & |k-l| = 1 \mod N \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

Thus, the Hamiltonian matrix is given by $H = \frac{P^2}{2m} + \text{diag}(V(x_k))$, retaining the sparsity properties of the discretized kinetic term.

2.4 Lanczos Algorithm for Eigenvalue Problems

The Lanczos algorithm is an iterative method for sparse, large-scale Hermitian matrices. The process begins with a randomly initialized normalized vector v_0 and iteratively generates an orthonormal basis for the Krylov subspace $\mathcal{K}_k(\hat{H}, v_0) = \text{span}\{v_0, \hat{H}v_0, \ldots, \hat{H}^{k-1}v_0\}$, thereby reducing $\hat{H}$ to a triadiagonal matrix T_k. The eigenvalues of this tridiagonal matrix approximate the low-lying eigenvalues of the original Hamiltonian, and full re-orthogonalization is applied at each iteration to maintain numerical stability [4].

The efficiency of the Lanczos method arises from its reliance on matrix-vector products rather than explicit matrix storage—a critical advantage for large-scale quantum systems where the Hamiltonian is sparse. This approach generates a sequence of vectors using a three-term recurrence relation, resulting in a tridiagonal matrix whose diagonal and off-diagonal elements capture the recurrence coefficients. For systems like the harmonic oscillator, the kinetic energy operator contributes a natural tridiagonal structure (via finite differences), and the potential energy operator remains diagonal, which allows rapid computation of $\hat{H}\psi$. This combination preserves the sparsity of the overall Hamiltonian, ensuring that each iteration remains computationally tractable and can be executed in $O(N)$ operations.

Convergence of the Lanczos algorithm is typically faster for extremal eigenvalues of T_k, but due to rounding errors and loss of orthogonality in finite-precision arithmetic, periodic re-orthogonalization is essential. This step ensures that the generated Lanczos vectors remain mutually orthogonal, preventing spurious convergence and improving the reliability of the computed eigenvalues. Balancing numerical stability with computational efficiency is a key challenge, particularly in parallel implementations.

Parallelization of the Lanczos algorithm is achieved through domain decomposition. The wave function ψ is partitioned across multiple MPI processes, with each process responsible for a segment of the spatial grid. For the harmonic oscillator, $\hat{H}\psi$ involves local stencil operations (nearest-neighbour sums), ideal for distributed computing. Additionally, OpenMP threading accelerates local operations, such as vector updates and inner products. While the orthogonalization step is inherently sequential, it leverages distributed linear algebra libraries to mitigate bottlenecks.

In practical implementations, the Lanczos algorithm [5] offers an efficient and scalable framework for extracting a few low-lying eigenstates from large, sparse Hamiltonians. Its iterative nature, combined with advanced parallelization techniques, makes it especially suitable for high-performance quantum mechanical simulations where only a narrow subset of the full spectrum is required. This focus on low-lying eigenstates is particularly advantageous in studies of ground-state properties requiring only a few eigenvalues and eigenvectors with low-energy excitations in quantum systems.

Chapter 3

Numerical Methods and Parallelization

3.1 Discretization Scheme

The spatial domain is discretized as follows:

- **Grid points:** $x \in \{-Ma, \ldots, -a, 0, a, \ldots, Ma\}$ with $N = 2M + 1$ points,
- **Lattice spacing:** $a = L/N$,
- **Momentum discretization:** $p_n = \frac{2\pi n}{L}, \quad n \in -M, \ldots, +M$,
- **Periodic boundary conditions:** $\phi(x + L) = \phi(x)$.

Two discretization methods to approximate the kinetic energy operator $\hat{p}^2$, are:

1. **Finite Difference Discretization:** The second derivative is approximated as

$$[\hat{p}^2\psi]_i = \frac{\hbar}{a^2}\left(2\psi(x_i) - \psi(x_{i-1}) - \psi(x_{i+1})\right), \tag{3.1}$$

   ensuring periodic boundary conditions via modular indexing.

2. **Spectral Derivative:** The second derivative is evaluated via a discrete Fourier transform, with matrix elements

$$P^2_{kl} = \frac{\hbar^2}{N}\sum_p p^2 \cos\left(p(x_k - x_l)\right), \tag{3.2}$$

   which reproduces the continuum eigenvalues with minimal discretization error.

3.2 Lanczos Algorithm Implementation

The Lanczos algorithm is implemented in Python language [1, 7]. The key steps are:

- **Initialization:** Begin with a random normalized vector.
- **Iteration:** Apply the Hamiltonian to generate a new Krylov basis vector. At each iteration, compute the diagonal (α) and off-diagonal (β) coefficients while performing full re-orthogonalization to mitigate numerical instabilities.

- **Tridiagonalization:** Construct the tridiagonal matrix, whose eigenvalues approximate those of $\hat{H}$.
- **Eigenvalue Extraction:** Use standard routines (e.g., from `scipy.sparse.linalg` import `eigsh`, use `ARPACK`) to solve the reduced eigenvalue problem.

The iterative process generates an orthonormal basis $\{v_j\}$ that tridiagonalizes the Hermitian matrix $\hat{H}$ while preserving its spectral properties. Re-orthogonalization at each

Algorithm 1 Lanczos Algorithm

procedure LANCZOS ITERATION($m, n, a, rank, num\_procs, comm$)
  $sizes \leftarrow$ DISTRIBUTE_ELEMENTS(n, num_procs)
  $local\_n \leftarrow sizes[rank]$
  $starts \leftarrow [\sum_{i=0}^{p-1} sizes[i]$ for $p = 0, \ldots, num\_procs - 1]$
  $start\_idx \leftarrow starts[rank]$
  $M \leftarrow \lfloor (n-1)/2 \rfloor$ `# Precomputing local position array`
  $x\_local \leftarrow a \cdot \text{range}(start\_idx - M, start\_idx - M + local\_n)$
  $v \leftarrow$ zero matrix of size $(local\_n \times m)$ `# Orthonormal basis`
  $\beta \leftarrow$ zero vector of length $m - 1$ `# Off-diagonal elements`
  $\alpha \leftarrow$ zero vector of length m `# Diagonal elements`
  $v_1 \leftarrow$ random vector of size $local\_n$ `# Initialize starting vector`
  $v_1 \leftarrow$ NORMALIZE_VECTOR(v_1, $comm$)
  $v\_temp \leftarrow$ EXCHANGE_BOUNDARY_DATA(v_1, $local\_n$, $rank$, num_procs, $comm$)
  $w_1^p \leftarrow$ HV(v_temp, a, x_local, $comm$) `# First Lanczos iteration`
  $a_1 \leftarrow$ DOT_PRODUCT(w_1^p, v_1, $comm$)
  $w_1 \leftarrow w_1^p - a_1 \cdot v_1$
  $\alpha[0] \leftarrow a_1$
  $v[:, 0] \leftarrow v_1$
  for $j \leftarrow 1$ to $m - 1$ **do** `# Lanczos iteration loop`
    $\beta[j-1] \leftarrow$ GET_GLOBAL_NORM(w_1, $comm$)
    if $\beta[j-1] \approx 0$ **then**
      $v[:, j] \leftarrow$ NORMALIZE_VECTOR(random vector of size $local\_n$, $comm$)
    else
      $v[:, j] \leftarrow \frac{w_1}{\beta[j-1]}$
    $v \leftarrow$ MODIFIED_GRAM_SCHMIDT(v, $j+1$, $comm$) `# C kernel`
    if $j \bmod 10 = 0$ **then** `# Check orthogonality: every 10 iterations`
      CHECK_ORTHOGONALITY(v, $j+1$, $comm$)
    $v\_tp \leftarrow$ EXCHANGE_BOUNDARY_DATA($v[:, j]$, $local\_n$, $rank$, num_procs)
    $w_1^p \leftarrow$ HV(v_temp, a, x_local, $comm$) `# C-accelerated kernel`
    $\alpha[j] \leftarrow$ DOT_PRODUCT(w_1^p, $v[:, j]$, $comm$)
    $w_1 \leftarrow w_1^p - \alpha[j] \cdot v[:, j] - \beta[j-1] \cdot v[:, j-1]$
  if $rank = 0$ **then** `# root process`
    $T \leftarrow$ tridiagonal matrix with main diagonal α and off-diagonals β
    $eigenvalues \leftarrow$ Compute 10 smallest eigenvalues from T
  return $eigenvalues$

step counteracts rounding errors, ensuring the stability and accuracy of the computed eigenvalues.

3.3 Hybrid MPI+OpenMP Parallelization

For handling large-scale simulations efficiently, the spatial domain is partitioned among MPI processes, with each process handling a contiguous block of grid points, thus holding a segment of the full wave function. OpenMP threads accelerate local computations, while MPI coordinates data exchange and global operations. Key features include:

- **Domain Decomposition:** Each MPI process holds N/N_{proc} contiguous grid points.
- **Boundary Exchange:** Non-blocking MPI calls exchange ghost cells with neighbouring processes, ensuring that finite difference approximation for the Hamiltonian access adjacent grid points.
- **Collective Operations:** Global inner products and norms are computed via MPI reduction operations during the orthogonalization steps.
- **Overlap of Computation and Communication:** Non-blocking communications are overlapped with local computations, optimizing overall execution time.
- **OpenMP Acceleration:** Critical kernels, such as the Hamiltonian application and Modified Gram-Schmidt re-orthogonalization, are implemented in C with OpenMP directives to exploit thread-level parallelism.

A pseudocode for the distributed matrix-vector product using this hybrid model is:

Algorithm 2 Distributed Matrix-Vector Product in Hybrid MPI+OpenMP

procedure DISTRIBUTED_MATVEC(v_{local}, $local\_n$, $rank$, num_procs)
  $right\_rank \leftarrow (rank + 1) \mod num\_procs$
  $left\_rank \leftarrow (rank - 1) \mod num\_procs$
  `# Initiate non-blocking communication for ghost cells`
  MPI_ISEND($v_{local}[0]$, $destination = left\_rank$)
  MPI_ISEND($v_{local}[local\_n - 1]$, $destination = right\_rank$)
  MPI_IRECV($ghost\_left$, $source = right\_rank$)
  MPI_IRECV($ghost\_right$, $source = left\_rank$)
  MPI_WAITALL
  `# Build extended vector with ghost cells`
  Set $v_{ext}[1 : local\_n] \leftarrow v_{local}$
  Set $v_{ext}[0] \leftarrow ghost\_left$, and $v_{ext}[local\_n + 1] \leftarrow ghost\_right$
  `# Apply Hamiltonian operator using OpenMP-accelerated kernel`
  $w_{local} \leftarrow$ HV(v_{ext}, a, x_{local}, $local\_n$)
  return w_{local}

In our implementation, the Python code (using `mpi4py`) manages the MPI tasks while delegating computationally intensive parts—such as the Hamiltonian application and the Modified Gram-Schmidt process—to C kernels optimized with OpenMP. This hybrid strategy enables us to scale to larger problem sizes and significantly reduces the execution time.

Chapter 4

Results: Strong Scaling Performance

4.1 Serial Implementation

The serial implementation employs the Lanczos algorithm to approximate the eigenvalues of a one-dimensional quantum Hamiltonian. The Hamiltonian operator is discretized using a finite difference approximation for the kinetic term in conjunction with a power-law potential. The algorithm is implemented in Python, with key routines—such as the Hamiltonian application and the Modified Gram-Schmidt reorthogonalization—optimized for efficiency. The spatial domain $x \in [-L/2, L/2]$ is discretized into $N = 2M + 1$ grid points with lattice spacing $a = L/N$. The corresponding momentum grid is defined by $p_n = \frac{2\pi n}{L}$ for $n = -M, \ldots, +M$, with periodic boundary conditions enforced.

A random normalized vector initiates the Lanczos process, which iteratively constructs an orthonormal basis of the Krylov subspace. The Hamiltonian matrix is effectively represented as a sparse tridiagonal matrix: the diagonal elements (α) capture the potential energy, while the off-diagonal elements (β) correspond to the kinetic energy. Full reorthogonalization is performed at each iteration via Modified Gram-Schmidt to ensure numerical stability. The resulting tridiagonal matrix is then diagonalized using routines from (`scipy.sparse.linalg.eigsh`) to extract the ten smallest eigenvalues, serving as a foundational benchmark for assessing the algorithm's convergence and accuracy.

We focus on the quantum harmonic oscillator, ($V(x) = \frac{1}{2}x^2$), using the following baseline parameters:

- **Grid Resolution:** $N = 153$ points over $L = 25$, yielding a lattice spacing $a = L/N \approx 0.163$,
- **Krylov Subspace Size:** Initially, $m = N$ (full orthogonalization); subsequent tests with fixed m and increasing N,
- **Natural Units:** $\hbar = m = \omega = 1$, with analytical eigenvalues given by $E_n = n + \frac{1}{2}$.

For the baseline case ($m = N = 153$), the Lanczos algorithm converges rapidly. The computed eigenvalues (presented in Table 4.1) agree closely with the analytical solutions, with relative errors below approximately 1.7% for the ten smallest eigenvalues, i.e., in the order of 10^{-3} to 10^{-2}. Residual norms are monitored at each iteration to ensure convergence and stability. To further investigate the method's performance, we consider a fixed Krylov subspace size m at its baseline value (153) and increase the total number of grid points N (i.e., $N = 153, 307, 613,$ etc.). This analysis reveals how discretization effects influence the deviation of the computed eigenvalues from the analytical values.

Table 4.1: Computed Eigenvalues and Relative Errors (Serial Implementation)

n	$E_n^{Computed}$	*Relative Error*
0	0.4992	−0.17%
1	1.4958	−0.28%
2	2.4891	−0.44%
3	3.4790	−0.60%
4	4.4655	−0.77%
5	5.4486	−0.93%
6	6.4283	−1.10%
7	7.4045	−1.27%
8	8.3772	−1.44%
9	9.3465	−1.62%

Figure 4.1 illustrates the relative error for the lowest eigenvalue as a function of the grid resolution. As N increases while m remains fixed, the relative error gradually deviates from the analytical solution, indicating the impact of discretization and the limitation of the fixed Krylov subspace size in capturing the complete spectral properties of the Hamiltonian.

Figure 4.1: Relative Error vs. Grid Resolution

4.2 Parallel Implementation

4.2.1 Parallel Implementation on Google Colab

The parallel implementation on Google Colab leverages MPI-based domain decomposition for the Lanczos algorithm [6]. The code, written in Python with `mpi4py`, distributes the

computational domain across MPI processes and employs non-blocking communication to exchange ghost cells between neighbouring processes. The Hamiltonian operator is evaluated using finite differences for the kinetic term and a power-law formulation for the potential term.

For the performance evaluation, the algorithm was executed for a fixed physical domain ($L = 25.0$), a constant Krylov subspace size ($m = 90$) while varying the number of spatial grid points N (specifically, $N = 513, 1025,$ and 2049). Experiments were conducted on three different runtime types available on Colab: CPU, GPU, and TPU. Table 4.2 summarizes representative execution times, and Figure 4.2 illustrates the trend in execution time as a function of grid resolution.

Table 4.2: Execution Times (*seconds*) on Google Colab for varying Grid Resolutions (N) and Runtime Types

N	*CPU Time*	*GPU Time*	*TPU Time*
513	11.8043	9.9576	4.0810
1025	14.4722	10.8169	4.7400
2049	15.2353	11.2835	6.4638

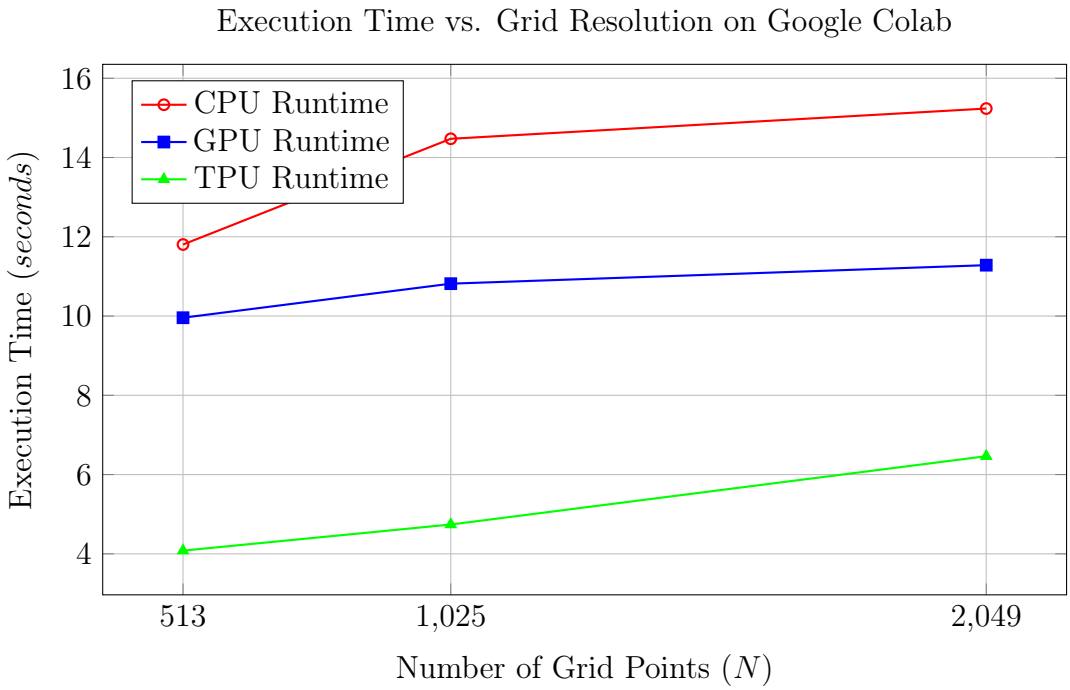

Figure 4.2: Execution time as a function of the grid resolution for three different runtime types on Google Colab. The TPU runtime achieves the fastest performance, followed by the GPU and CPU runtimes.

The results indicate that for a constant physical domain and fixed Krylov subspace size, increasing the number of grid points (i.e., refining the spatial resolution) leads to a moderate increase in execution time on the CPU and GPU. On the other hand, TPU exhibits a more pronounced impact at the highest resolution. Notably, the TPU—with its 8−core accelerator architecture (totaling 96 cores)—significantly outperforms the CPU (2 cores) and GPU (2 cores) implementations. These observations provide valuable insights

into the performance characteristics of the MPI-only parallel implementation across the different hardware accelerators available on Google Colab.

4.2.2 Parallel Implementation on Stromboli Cluster

The parallel implementation on the Stromboli Cluster employs a hybrid MPI+OpenMP approach to accelerate the Lanczos algorithm for one-dimensional quantum mechanics. In this implementation, the computational domain is partitioned across MPI processes. At the same time, performance-critical kernels—namely, the Hamiltonian operator and the Modified Gram–Schmidt orthogonalization—are optimized in C using OpenMP directives. The MPI-based domain decomposition efficiently manages inter-process communication (including ghost cell exchanges), and the OpenMP kernels provide additional acceleration on each node.

The code is organized in the file `matrix_qm_parallel_mpi_omp.py`, which interfaces with the optimized C/OpenMP library (`lanczos_openmp.so`). A substantial scaling study was performed on the Stromboli Cluster by fixing the total problem size (grid points N) and the Krylov subspace size m while varying the number of MPI processes from 2 to 64. The SLURM script (`submit_strong.sh`) automates job submission with varying process counts. Execution times were recorded for several grid sizes (e.g., $N = 32769, 65537, \ldots, 4194305$) and different values of m (e.g., $m = 30, 45, 60,$ and 75).

Figure 4.3 shows an illustrative strong scaling plot for the representative case with $m = 60$ and $N = 524289$. Here, the measured speedup is defined as the ratio of the execution time at 2 processes to that at a higher process count. For comparison, the ideal speedup—assumed to be $p/2$ (based on the 2−process baseline)—is also plotted. The results indicate that the measured speedup closely approaches the ideal trend, with only minor deviations at higher processor counts, primarily due to increased communication overhead and node-level load imbalance.

To better understand the sensitivity of scalability to the Krylov subspace size m, Figure 4.4 compares strong scaling performance at a fixed problem size $N = 131073$ across varying $m = 30, 45, 60,$ and 75. The results reveal diverse scaling trends:

- For $m = 30$, the speedup curve shows early saturation and deviates significantly from ideal scaling, likely due to insufficient computational workload per process,
- For $m = 45$ and $m = 60$, the scaling improves and aligns more closely with the ideal trend,
- For $m = 75$, the measured speedup briefly surpasses the ideal speedup—possibly due to favourable cache effects or temporal variability in node performance—but ultimately flattens.

Complementarily, Figure 4.5 presents strong scaling for fixed $m = 30$ over a range of increasing grid sizes N. Larger problem sizes exhibit better scalability, as expected, due to an increased compute-to-communication ratio. The lowest grid ($N = 32769$) exhibits early saturation and significant deviation from the ideal trend, whereas for larger grids ($N = 131073$ and above), the speedup trend approaches the ideal linear scaling.

A consequential aspect of the hybrid implementation is the efficient exchange of boundary data among MPI processes. The function `exchange_boundary_data` performs non-blocking communication to exchange ghost cells between neighbouring processes. Each process maintains a local subdomain augmented with ghost cells at the left and

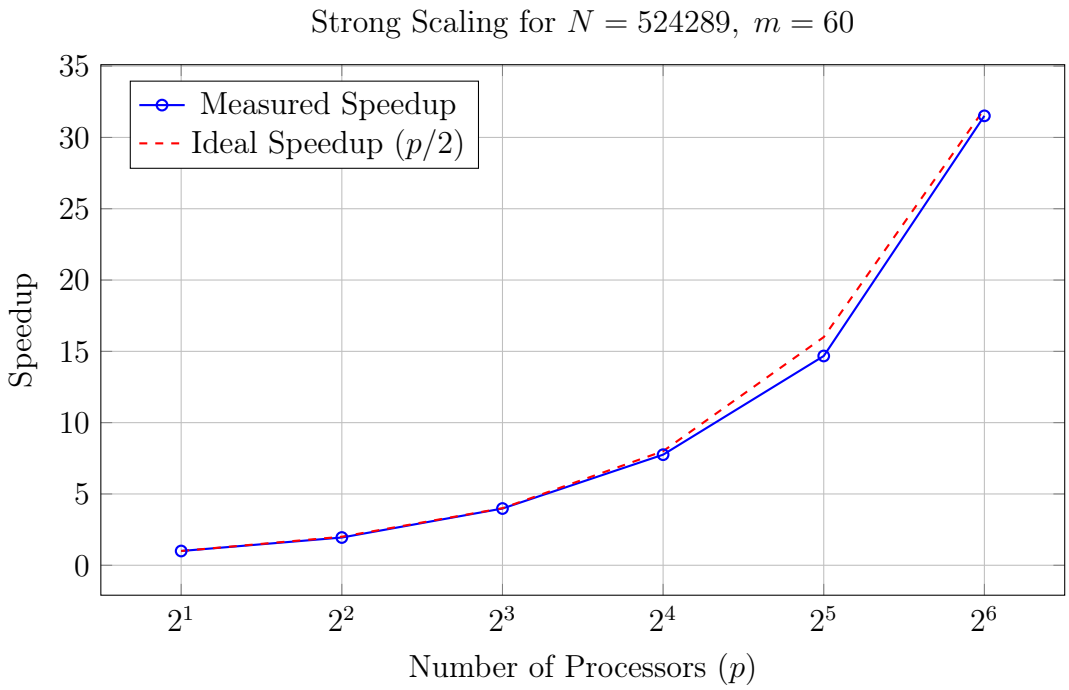

Figure 4.3: Strong scaling performance for $m = 60$ and $N = 524289$ on the Stromboli Cluster

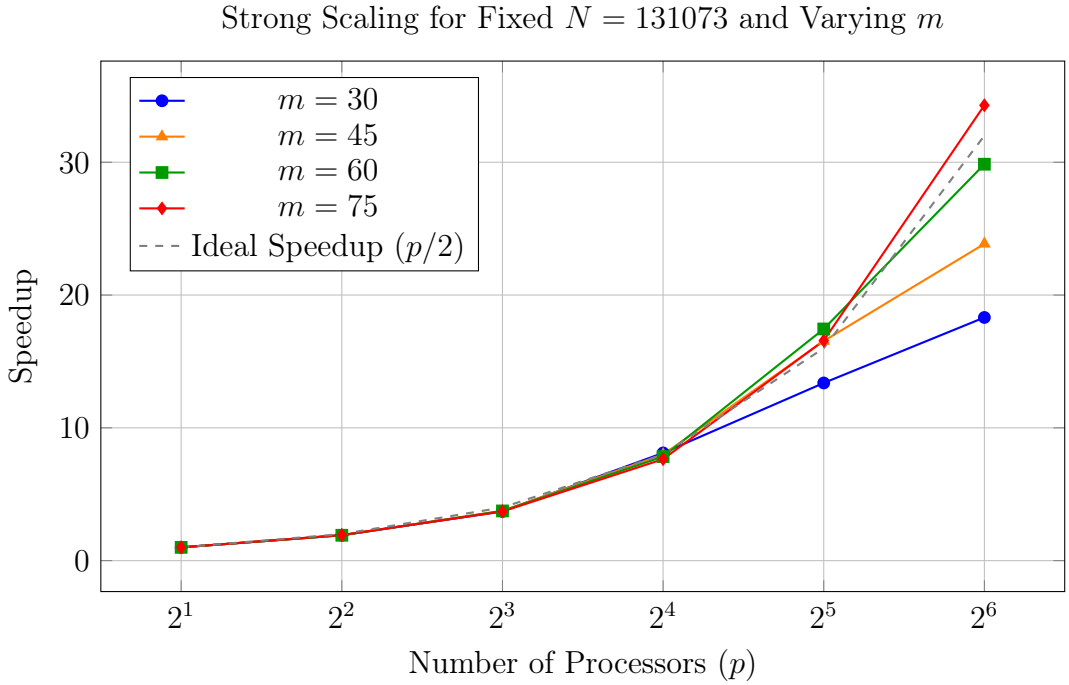

Figure 4.4: Speedup comparison for different values of m at fixed $N = 131073$

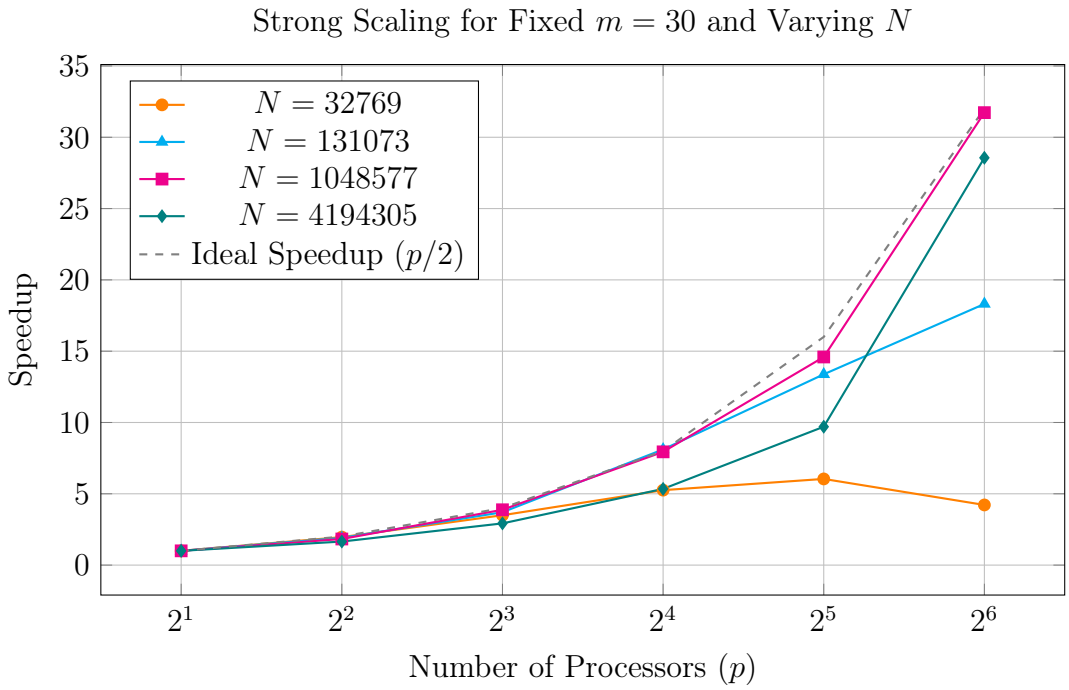

Figure 4.5: Speedup comparison for different grid sizes N at fixed $m = 30$

right boundaries. These ghost cells store data from adjacent processes to ensure that finite-difference operations—used in the Hamiltonian operator—can be executed without interruption. This boundary exchange is critical for maintaining data consistency and achieving high parallel efficiency. In particular, for a large computational domain and for the inter-process communication where overhead becomes significant.

To illustrate this mechanism, Figure 4.6 depicts a schematic of the domain decomposition. Each row represents the local subdomain of an MPI process (blue cells) sandwiched by ghost cells (orange cells). Arrows indicate the non-blocking communication flow which transfers boundary information between neighbouring processes. Blue arrows represent data sent from the first local cell of a process (e.g., Process A or B) to the right ghost cell of its neighbouring process, and red arrows indicate data sent from the last local cell (i.e., the last blue cell) to the left ghost cell of the adjacent process.

These scaling studies indicate that the overall parallel performance depends critically on the Krylov subspace size m and the total number of grid points N. Smaller m or N may lead to suboptimal scaling due to limited per-processor workload, conversely, larger values of m and N help sustain near-ideal or even superlinear speedup due to improved parallel efficiency, cache utilization, and reduced relative communication overhead. Overall, these results validate the effectiveness of the hybrid MPI+OpenMP strategy for large-scale quantum simulations [3] on modern HPC architectures.

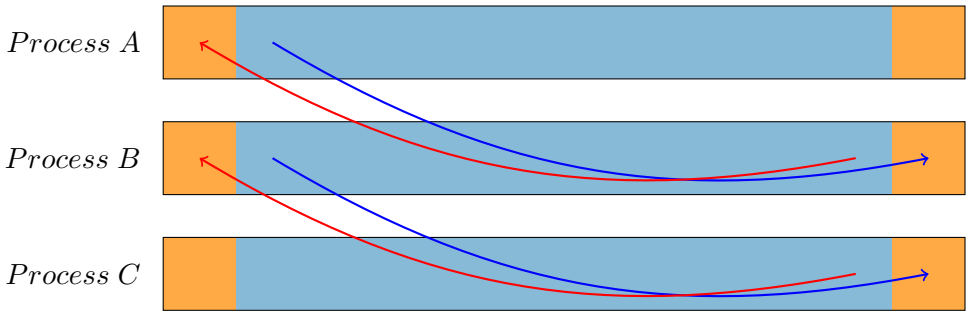

Figure 4.6: Schematic illustration of boundary exchange in the hybrid MPI+OpenMP implementation. Each MPI process holds its local subdomain (blue cells) and ghost cells (orange) at the boundaries. Blue arrows represent data sent from the first local cell of a process (e.g., Process A or B) to the right ghost cell of the neighbouring process, while red arrows depict data sent from the last local cell (i.e., the last blue cell) to the left ghost cell of the adjacent process.

Chapter 5

Conclusion and Outlook

In this work, we have developed and rigorously validated a numerical framework for solving quantum mechanical eigenvalue problems using a matrix formulation and the Lanczos algorithm. By discretizing the Schrödinger equation for one-dimensional systems, we effectively reduced the continuous problem into a large, sparse matrix eigenvalue problem. This approach was successfully applied to benchmark systems—such as the quantum harmonic oscillator—where the computed eigenvalues show excellent agreement with the analytical solutions.

The serial implementation based on a finite difference discretization combined with full reorthogonalization via the Modified Gram–Schmidt process delivers a robust baseline. Detailed convergence studies demonstrate that the Lanczos algorithm converges rapidly with minimal relative error, confirming its accuracy and efficiency even for large matrix dimensions. Building on this foundation, we extended the numerical framework to a parallel implementation using MPI for inter-process communication. On the Google Colab platform, the MPI-only version displayed significant execution time improvements for varying grid resolutions. The hybrid MPI+OpenMP implementation on the Stromboli Cluster achieved nearly ideal strong scaling performance across a range of problem sizes and Krylov subspace dimensions.

A key innovation in our approach is the efficient management of ghost cell exchanges, which ensures consistency in finite-difference operations across process boundaries. Optimizations at both the algorithmic and code levels—such as utilizing OpenMP-accelerated kernels for the Hamiltonian operator and orthogonalization routines—were critical in achieving high computational performance on modern HPC architectures.

Outlook

The positive results obtained from our numerical framework suggest several promising directions for future research:

- **Extension to Higher Dimensions:** The current framework is limited to one-dimensional systems. Future work will extend these methods to two- and three-dimensional systems, which will require more advanced discretization strategies and shall pose additional computational challenges.

- **Enhanced Parallelization Strategies:** Further improvements in scalability could be achieved by exploring sophisticated load-balancing techniques, hybrid approaches

that combine MPI with GPU acceleration, and the emerging hardware paradigms integration.

- **Application to Complex Quantum Systems:** The matrix formulation and Lanczos algorithm can be adapted to more intricate potential landscapes and many-body quantum systems. This extension will be significant for studying quantum phase transitions, strongly correlated systems, and quantum dynamics.

- **Algorithmic Refinements:** Future work may incorporate adaptive techniques to dynamically adjust the Krylov subspace size and explore alternative reorthogonalization methods to improve accuracy and performance.

- **Software Integration and Benchmarking:** Integrating the present framework with established quantum mechanics simulation packages and benchmarking against state-of-the-art methods would further validate its performance and help identify further optimization opportunities.

In summary, our work lays a solid foundation for the numerical exploration of quantum systems through efficient matrix methods and scalable parallel implementations. The encouraging results provide a clear pathway for future research, which will broaden the applicability of these techniques to more complex physical systems and computational environments.

Acknowledgements: First and foremost, we express our sincere gratitude to advisor, *Dr. Tomasz Korzec*, for the opportunity to work on such an engaging project and for his invaluable guidance and continuous support throughout this project. We also acknowledge the Google Colab platform and the computational resources provided by the *Stromboli* cluster at Bergische Universität Wuppertal.

Appendices

Appendix A

Python/MPI Program Code Listings

A.1 Lanczos Algorithm Implementation

```
def Lanczos_algorithm(m, n, a, rank, num_procs, comm):

    # Domain decomposition: Splitting grid points across processes
    sizes = distribute_elements(n, num_procs)
    local_n = sizes[rank]
    starts = [sum(sizes[:i]) for i in range(num_procs)]
    start_idx = starts[rank]

    # Precomputing the local position array for current process
    M = (n - 1) // 2
    x_local = a * np.arange(start_idx - M, start_idx - M + local_n)

    # Initializing Lanczos vectors and coefficients
    v = np.zeros((local_n, m), dtype=np.float64)   # Orthonormal basis
    B = np.zeros(m - 1, dtype=np.float64)    # Off-diagonal elements
    alpha = np.zeros(m, dtype=np.float64)    # Diagonal elements

    # Initializing the starting vector (local part) and normalizing it
   globally
    v1 = np.random.randn(local_n)
    v1 = normalize_vector(v1, comm)

    # First Lanczos iteration: applying Hv using boundary exchange
    v_temp = exchange_boundary_data(v1, local_n, rank, num_procs, comm)
    w1_p = Hv(v_temp, a, x_local, comm)
    a1 = dot_product(w1_p, v1, comm)
    w1 = w1_p - a1 * v1
    alpha[0] = a1
    v[:, 0] = v1

    comm.Barrier()
    start_time = MPI.Wtime()

    # Lanczos iteration loop
    for j in range(1, m):
        # Computing the beta coefficient with global norm
        B[j - 1] = get_global_norm(w1, comm)

        # Handling the Lanczos breakdown
        if np.isclose(B[j-1], 0.0):
```

```
            # Restart if numerical breakdown occurs
            v[:, j] = normalize_vector(np.random.randn(local_n), comm)
        else:
            v[:, j] = w1 / B[j-1]

        # Function call: C-accelerated Modified Gram-Schmidt reorthogonalization
        v = modified_gram_schmidt(v, j + 1, comm)

        # Applying the Hamiltonian operator (with boundary communication)
        v_temp = exchange_boundary_data(v[:, j], local_n, rank, num_procs, comm)
        w1_p = Hv(v_temp, a, x_local, comm)
        alpha[j] = dot_product(w1_p, v[:, j], comm)
        w1 = w1_p - alpha[j]*v[:, j] - B[j - 1]*v[:, j - 1]

    comm.Barrier()
    iteration_time = MPI.Wtime() - start_time

    # Root process (rank 0) solves tridiagonal system and evaluate eigenvalues
    if rank == 0:
        T = np.diag(alpha) + np.diag(B, 1) + np.diag(B, -1)
        eigenvalues, _ = eigsh(T, k=10, which='SA')
    else:
        eigenvalues = None

    return eigenvalues
```

A.2 Loading C/OpenMP library & defining function signatures

```
# Loading C/OpenMP library for accelerated kernels
lib = CDLL('./lanczos_openmp.so')
# Defining C function signatures
lib.Hv.argtypes = [
    POINTER(c_double),  # Input vector v with ghost cells (for boundary exchange)
    c_double,           # Grid spacing 'a'
    POINTER(c_double),  # Local position array
    c_int,              # Local vector size (number of grid points in current process)
    POINTER(c_double)   # Output vector for the Hamiltonian result
]
lib.Hv.restype = None

lib.modified_gram_schmidt.argtypes = [
    POINTER(c_double),  # Flattened matrix data for Lanczos vectors
    c_int,              # Number of rows (local grid points)
    c_int               # Number of columns (current iteration count 'k')
]
lib.modified_gram_schmidt.restype = None
```

A.3 MPI Helper Functions

```python
def distribute_elements(N, num_procs):
    """
    Distributes grid points evenly across MPI processes
    """
    base, rem = divmod(N, num_procs)
    return [base + 1 if i < rem else base for i in range(num_procs)]

def exchange_boundary_data(v, local_n, rank, size, comm):
    """
    Performs non-blocking communication to exchange ghost cell data with neighboring processes
    """
    right_rank = (rank + 1) % size
    left_rank = (rank - 1) % size

    # Preparing the send/receive buffers
    send_buf = np.array([v[0], v[-1]], dtype=np.float64)
    recv_buf = np.empty(2, dtype=np.float64)

    # Non-blocking communications
    comm_time = -MPI.Wtime()
    reqs = [
        comm.Isend(send_buf[0:1], dest=left_rank),
        comm.Isend(send_buf[1:2], dest=right_rank),
        comm.Irecv(recv_buf[0:1], source=right_rank),
        comm.Irecv(recv_buf[1:2], source=left_rank)
    ]
    MPI.Request.Waitall(reqs)
    comm_time += MPI.Wtime()

    # Building the extended local vector with ghost cells at boundaries
    v_ext = np.empty(local_n + 2, dtype=np.float64)
    v_ext[1:-1] = v   # Local data
    v_ext[0] = recv_buf[0]  # Left ghost cell
    v_ext[-1] = recv_buf[1] # Right ghost cell

    if rank == 0:
        print(f"Comm time per step: {comm_time:.6f}s")
    return v_ext

def normalize_vector(v, comm):
    """Normalizes the vector using a global norm computed via MPI"""
    return v / get_global_norm(v, comm)

def get_global_norm(v, comm):
    """Computes the L2 norm of a vector across all MPI processes"""
    local_sq = np.sum(v**2)
    global_sq = comm.allreduce(local_sq, op=MPI.SUM)
    return np.sqrt(global_sq)

def dot_product(v, u, comm):
    """Computes the global dot product of two vectors across MPI processes"""
    local_dot = v @ u
    return comm.allreduce(local_dot, op=MPI.SUM)
```

A.4 Python wrappers for calling C routines

```python
def Hv(v, a, x_local, comm):
    """
    Applies the Hamiltonian operator using the optimized C/OpenMP
    kernel
    """
    local_n = len(x_local)
    v_ptr = v.ctypes.data_as(POINTER(c_double))
    x_ptr = x_local.ctypes.data_as(POINTER(c_double))
    result = np.zeros(local_n, dtype=np.float64)
    result_ptr = result.ctypes.data_as(POINTER(c_double))

    # Calling C function via ctypes interface
    lib.Hv(v_ptr, c_double(a), x_ptr, c_int(local_n), result_ptr)
    return result

def modified_gram_schmidt(matrix, k, comm):
    """
    Applies the C-accelerated Modified Gram-Schmidt orthogonalization
    """
    local_n, m = matrix.shape
    matrix_flat = np.ascontiguousarray(matrix[:, :k].flatten())
    matrix_ptr = matrix_flat.ctypes.data_as(POINTER(c_double))

    # Calling C function via ctypes interface
    lib.modified_gram_schmidt(matrix_ptr, c_int(local_n), c_int(k))

    # Reshaping back to original dimensions
    matrix[:, :k] = matrix_flat.reshape((local_n, k))
    return matrix
```

Appendix B

C/OpenMP Program Code Listings

B.1 C Kernel for Hamiltonian Application

```c
// Function: Hv
void Hv(double* v, double a, double* x_local, int local_n, double* result) {
    const double hbar2 = 1.0;
    const double coeff = hbar2 / (2 * a * a);
    int j;

    // Kinetic Energy: Central Difference approximation
    #pragma omp parallel for default(none) shared(v, result, local_n) private(j)
    for (j = 0; j < local_n; j++) {
        result[j] = coeff * (v[j] + v[j + 2] - 2 * v[j + 1]);
    }

    // Potential Energy: Harmonic oscillator potential
    #pragma omp parallel for default(none) shared(v, x_local, result, local_n) private(j)
    for (j = 0; j < local_n; j++) {
        result[j] += 0.5 * x_local[j] * x_local[j] * v[j + 1];
    }
}
```

B.2 C Kernel for Modified Gram-Schmidt

```c
// Function: modified_gram_schmidt
void modified_gram_schmidt(double* matrix, int n, int k) {
    int col, row, other_col;
    for (col = 0; col < k; col++) {
        double norm = 0.0;
        // Computing the norm of the current column vector
        #pragma omp parallel for reduction(+:norm) default(none) shared(matrix, n, col) private(row)
        for (row = 0; row < n; row++) {
            norm += matrix[row + col * n] * matrix[row + col * n];
        }
        norm = sqrt(norm);

        // If norm is too small, then reinitializing to the identity basis (avoiding division by zero)
```

```
14          if (norm < 1e-15) {
15              #pragma omp parallel for default(none) shared(matrix, n,
    col) private(row)
16              for (row = 0; row < n; row++) {
17                  matrix[row + col * n] = (row == col % n) ? 1.0 : 0.0;
18              }
19              norm = 1.0;
20          }
21
22          // Normalizing the column vector
23          #pragma omp parallel for default(none) shared(matrix, n, col,
    norm) private(row)
24          for (row = 0; row < n; row++) {
25              matrix[row + col * n] /= norm;
26          }
27
28          // Orthogonalizing subsequent columns against the current one
29          for (other_col = col + 1; other_col < k; other_col++) {
30              double dot = 0.0;
31              #pragma omp parallel for reduction(+:dot) default(none)
    shared(matrix, n, col, other_col) private(row)
32              for (row = 0; row < n; row++) {
33                  dot += matrix[row + other_col * n] * matrix[row + col *
     n];
34              }
35              #pragma omp parallel for default(none) shared(matrix, n,
    col, other_col, dot) private(row)
36              for (row = 0; row < n; row++) {
37                  matrix[row + other_col * n] -= dot * matrix[row + col *
     n];
38              }
39          }
40      }
41  }
```

Appendix C

Shell Script Listings

C.1 Shell Script for Job Submission

```
#!/bin/bash
#SBATCH --partition=compute2011

N=$((2**20 + 1))  # 1,048,577 grid points (odd for symmetry)
m=60      # Krylov subspace size

OUTPUT_DIR="strong_scaling_results"
CORES_PER_NODE=24 # Stromboli cluster configuration

# Looping over different process counts for strong scaling study
for P in 2 4 8 16 32 64; do
    if [ $P -eq 64 ]; then
        # For 64 MPI tasks, use 3 nodes; do not force tasks-per-node to let Slurm optimize allocation
        NODES=3
        TASKS_PER_NODE=""
    else
        # Calculating the number of nodes required and tasks per node for other cases
        NODES=$(( (P + CORES_PER_NODE - 1) / CORES_PER_NODE ))
        TASKS_PER_NODE=$(( (P + NODES - 1) / NODES ))
    fi

    echo "Submitting P=$P (nodes=$NODES)..."
    sbatch <<EOF
#!/bin/bash
#SBATCH --job-name=strong_P${P}
#SBATCH --output=strong_P${P}.log
#SBATCH --ntasks=${P}
#SBATCH --nodes=${NODES}
#SBATCH --partition=compute2011
#SBATCH --cpus-per-task=1

# OpenMP settings: using one thread per MPI process
export OMP_NUM_THREADS=1
export OMP_PLACES=cores
export OMP_PROC_BIND=close

# Disabling the internal multithreading for libraries to avoid oversubscription
```

```
export BLIS_NUM_THREADS=1
export MKL_NUM_THREADS=1
export OPENBLAS_NUM_THREADS=1
export NUMEXPR_NUM_THREADS=1
export VECLIB_MAXIMUM_THREADS=1

# Stability and performance flags
export MKL_SERIAL=yes
export MKL_DYNAMIC=FALSE
export OMP_DYNAMIC=FALSE

# Additional environment settings to ensure consistency
export SLURM_EAR_LOAD_MPI_VERSION="intel"
export MKL_DOMAIN_ALL=1
export MKL_THREADING_LAYER=sequential

# Executing the parallel Lanczos algorithm
srun python ./matrix_qm_parallel_mpi_omp.py --N ${N} --m ${m} --
    output_dir ${OUTPUT_DIR}
EOF
done
```

Bibliography

[1] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. Fourth. JHU Press, 2013. ISBN: 1421407949 9781421407944. URL: https://epubs.siam.org/doi/book/10.1137/1.9781421407944.

[2] David J. Griffiths. *Introduction to Quantum Mechanics (2nd Edition)*. 2nd. Pearson Prentice Hall, Apr. 2004. ISBN: 0131118927.

[3] Michael T. Heath. *Scientific Computing: An Introductory Survey*. 2nd. McGraw-Hill, 2002. ISBN: 9780071244893.

[4] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second. Society for Industrial and Applied Mathematics, 2002. DOI: 10.1137/1.9780898718027. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898718027.

[5] Cornelius Lanczos. "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators". In: *Journal of Research of the National Bureau of Standards* 45.4 (Oct. 1950), pp. 255–282. ISSN: 0160-1741 (print), 2376-5259 (electronic). DOI: https://doi.org/10.6028/jres.045.026.

[6] Peter S. Pacheco. *Parallel Programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. ISBN: 1558603395.

[7] William H. Press et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd. Cambridge University Press, 2007. ISBN: 0521880688.

[8] Jun John Sakurai and Jim Napolitano. *Modern Quantum Mechanics; 2nd ed.* San Francisco, CA: Addison-Wesley, 2011. URL: https://cds.cern.ch/record/1341875.

[9] Lloyd N. Trefethen. *Spectral Methods in MATLAB*. Society for Industrial and Applied Mathematics, 2000. DOI: 10.1137/1.9780898719598. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898719598.