Parallel RK4 Solver for Liouvillian Dynamics
Final Project Report

Jiahao Chen
CSE6230 – Fall 2025

December 13, 2025

Abstract

Contents

1 Introduction

1.1 Background and related work

Time integration of large linear ordinary differential equations is a central task in many areas of scientific computing [1,2]. In the theory of open quantum systems, the reduced density matrix $\rho(t)$ of a system interacting with an environment evolves according to a master equation of the form

$$\frac{d\rho}{dt} = \mathcal{L}[\rho],$$

where $\mathcal{L}$ is a linear Liouvillian superoperator that incorporates coherent Hamiltonian dynamics together with dissipative processes such as relaxation and dephasing [3,4]. For Markovian reservoirs, $\mathcal{L}$ often takes the Gorini–Kossakowski–Sudarshan–Lindblad form [5,6], while structured or strongly coupled environments lead to time-dependent or non-Markovian generators [3]. After vectorization, the evolution becomes a high-dimensional linear ODE

$$\frac{dr}{dt} = L(t)\, r,$$

where r contains the elements of ρ in some basis (e.g., $r = \mathrm{vec}(\rho)$) and $L(t)$ is a dense or structured matrix representation of the Liouvillian. Efficiently computing $r(t)$ is essential for predicting observables, steady states, relaxation dynamics, and transport properties in quantum optics, condensed matter physics, and quantum information science [3, 4].

Fourth-order Runge–Kutta (RK4) methods provide a robust balance of accuracy and stability for such linear systems [1, 2, 7]. However, when the Hilbert space dimension is N, the vectorized density matrix has dimension $D = N^2$, and the Liouvillian operator becomes a $D \times D$ matrix. The dominant cost of RK4 integration is the repeated application of $L(t)$ to intermediate stage vectors, which reduces to dense matrix–vector products [8]. For large D and long propagation times, these matrix–vector operations become the principal computational bottleneck, making Liouvillian dynamics a natural target for high-performance parallel computing [9].

Modern architectures combine multiple levels of parallelism. Shared-memory parallelism via OpenMP and distributed-memory parallelism via MPI provide complementary advantages, and hybrid models allow node-level and cluster-level scaling [9–11]. This project develops a high-performance RK4 solver for Liouvillian dynamics using multiple programming models (OpenMP, CUDA, MPI+OpenMP, and MPI+CUDA), with emphasis on accelerating the core matrix–vector kernel and understanding how performance depends on data decomposition and communication.

2 Methods

2.1 RK4 and serial implementation

We consider a linear ordinary differential equation of the form

$$\frac{dr}{dt} = L(t)\, r,$$

where $r(t) \in \mathbb{R}^D$ (or $\mathbb{C}^D$) is the vectorized density matrix and $L(t) \in \mathbb{R}^{D\times D}$ is the matrix representation of the Liouvillian. In the serial implementation we focus first on the time-independent

case $L(t) = L$, which already captures the main computational structure and simplifies verification. The extension to a time-dependent $L(t)$ only requires evaluating a different matrix at each time step or stage.

The state vector r is stored as a contiguous array of length D, and the matrix L is stored as a dense array of size $D \times D$. In the reference code we use a row-major layout, so that $L[i, j]$ is stored at index `i * D + j`. The dominant kernel is the matrix–vector product

$$y = L\,x,$$

which is implemented as a double loop over rows i and columns j with an inner accumulation for each row.

One time step of the classical fourth-order Runge–Kutta method is implemented as follows. Given r_n at time t_n and step size Δt, we compute

$$\begin{aligned} k_1 &= f(t_n, r_n) = L r_n, \\ k_2 &= f\big(t_n + \tfrac{\Delta t}{2}, r_n + \tfrac{\Delta t}{2} k_1\big) = L\big(r_n + \tfrac{\Delta t}{2} k_1\big), \\ k_3 &= f\big(t_n + \tfrac{\Delta t}{2}, r_n + \tfrac{\Delta t}{2} k_2\big) = L\big(r_n + \tfrac{\Delta t}{2} k_2\big), \\ k_4 &= f\big(t_n + \Delta t, r_n + \Delta t\, k_3\big) = L\big(r_n + \Delta t\, k_3\big), \\ r_{n+1} &= r_n + \frac{\Delta t}{6}\big(k_1 + 2k_2 + 2k_3 + k_4\big). \end{aligned}$$

In the code, the right-hand side function $f(t, r)$ is implemented by the matrix–vector kernel. The vectors $k_1, \ldots, k_4$ and temporary stage states are stored as additional arrays of length D that are reused at every step.

2.2 Serial implementation details

The main serial driver allocates memory for L and the vectors, initializes L and the initial condition $r(0)$, and then advances the solution from $t = 0$ to a final time T using $N_{\text{steps}} = T/\Delta t$ uniform steps. For verification we use diagonal or block-diagonal choices of L for which analytical solutions are available, and we measure the error

$$\varepsilon(T) = \|r_{\text{num}}(T) - r_{\text{exact}}(T)\|_2$$

as a function of Δt to confirm fourth-order convergence (when using a Δt sweep) and to establish a trusted serial reference against which all parallel implementations are compared.

2.3 Parallel implementation details – Row-wise decomposition

Our parallelization follows a simple one-dimensional row-wise decomposition of the global matrix and state vector.

We assume an operator $L \in \mathbb{C}^{D \times D}$ and a state vector $r \in \mathbb{C}^D$ (e.g., $r = \text{vec}(\rho)$ for a density matrix ρ). The global row index set $\{0, \ldots, D-1\}$ is partitioned into contiguous blocks, and each worker is assigned one block of rows. Concretely, worker w owns a subset of rows

$$i \in \mathcal{I}_w \subset \{0, \ldots, D-1\},$$

and stores

- the corresponding block of the matrix, $L^{(w)}$, containing rows $L_{i,:}$ for $i \in \mathcal{I}_w$;

- the corresponding segment of the state, $r^{(w)}$, containing entries r_i for $i \in \mathcal{I}_w$.

The Runge–Kutta scheme requires repeated matrix–vector products of the form

$$k_\ell = L\, v_\ell,$$

where the stage input vectors v_ℓ are combinations of r and the previous stage vectors k_1, k_2, k_3. For a dense matrix, each output component $k_{\ell,i}$ depends on all entries of the input vector, so every worker needs access to the full stage vector v_ℓ, but only has to compute its own rows.

Each RK4 stage proceeds in two conceptual steps:

1. Global vector assembly: all workers contribute their local segment of the current stage input (for example, $r^{(w)} + \frac{\Delta t}{2} k_1^{(w)}$), and a collective communication step is used to assemble a full copy of the vector $v_\ell \in \mathbb{C}^D$ on every worker.

2. Local matrix–vector product: given v_ℓ, worker w computes only its own portion of the stage vector,

$$k_\ell^{(w)}(i) = \sum_{j=0}^{D-1} L_{ij}\, v_\ell(j), \quad i \in \mathcal{I}_w,$$

using its local block $L^{(w)}$.

All vector updates in RK4, such as forming the stage inputs ($r + \frac{\Delta t}{2} k_1$, $r + \frac{\Delta t}{2} k_2$, $r + \Delta t\, k_3$) and the final update

$$r \leftarrow r + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$

are performed locally on each worker's segment $r^{(w)}$, without changing the row ownership. Thus, the global row partitioning remains fixed throughout the time evolution: each worker always updates the same subset of indices, and the only communication required per stage is the global exchange of the current stage input vector.

This description is independent of the concrete parallel programming model: whether workers correspond to MPI ranks, GPU devices, threads, or a hybrid configuration, they all follow the same pattern of row-wise data partitioning, global vector exchange per stage, and local matrix–vector computation.

2.4 Other parallel strategy

Row-wise vs. column-wise decomposition. Our implementation distributes the system matrix $L \in \mathbb{R}^{D\times D}$ by contiguous *rows* across p workers. In each RK4 stage, rank k owns $L_k \in \mathbb{R}^{D_k \times D}$ and the corresponding state segment $r_k \in \mathbb{R}^{D_k}$, but must access the full vector $r \in \mathbb{R}^D$ to form $k_k = L_k r$. This induces a collective exchange of the vector (e.g., `MPI_Allgatherv`) at every stage. An alternative is a *column-wise* partition, where rank k owns $L^{(k)} \in \mathbb{R}^{D\times D_k}$ and r_k; each rank computes a partial contribution $y^{(k)} = L^{(k)} r_k \in \mathbb{R}^D$ and the global product is obtained by summation (ideally via `MPI_Reduce_scatter` to return only the local segment). For a dense square matrix, both decompositions have the same arithmetic cost per matvec, $\Theta(D^2/p)$, and move $\Theta(D)$ vector data per stage in the dominant collective, so the asymptotic communication/computation balance is comparable. The primary practical difference is memory locality: with row-major storage, a row-wise decomposition streams contiguous memory in the innermost loop, whereas a column-wise decomposition tends to access strided data unless the local block is stored/transposed to match column traversal.

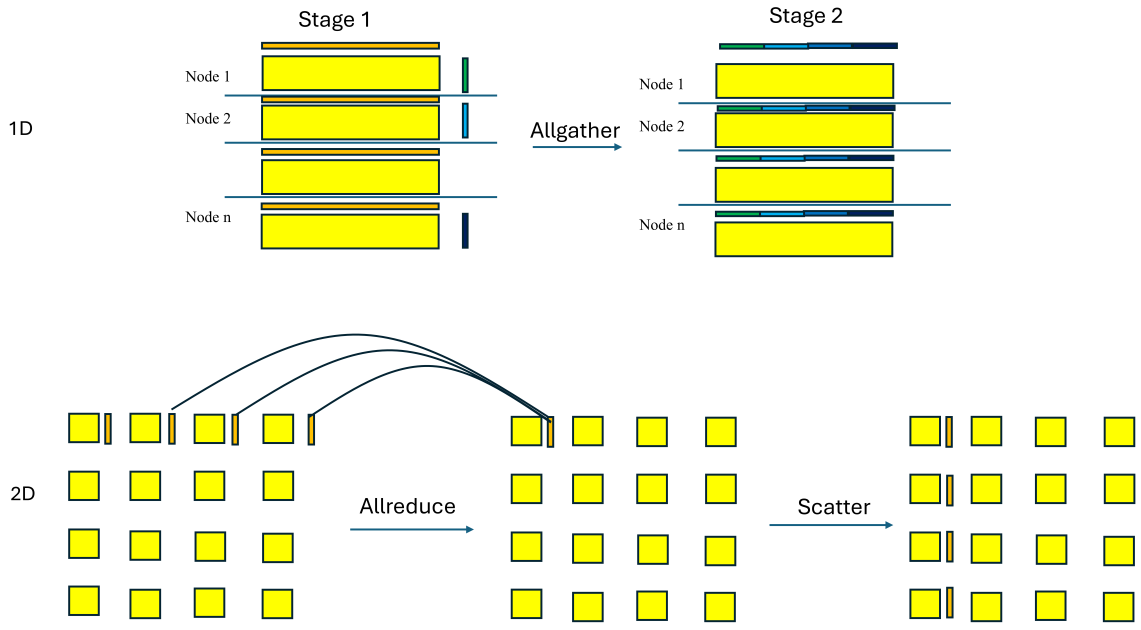

Figure 1: Data Distribution Communication scheme of 1D and 2D RK4 steps across distributed nodes. 1D uses allgather and collect a full vector on each node. 2D uses allreduce to reduce the local vector from the row and then scatter it to the same column.

1D vs 2D decomposition In an RK4 integrator each time step performs four distributed matrix–vector products, so the communication pattern of the matvec dominates the per-step communication cost. With a 1D row decomposition over p ranks, each rank owns approximately n/p rows of L but needs the full stage input vector $v_\ell \in \mathbb{R}^n$ to compute $k_\ell = Lv_\ell$, which typically forces one `MPI_Allgather` (or `MPI_Allgatherv`) per RK4 stage; in the standard α–β model this gives $T_{\text{1D,stage}} \approx \alpha \log p + \beta(nb)$ (where b is the bytes per element), hence $T_{\text{1D,step}} \approx 4(\alpha \log p + \beta nb)$, i.e., each rank effectively communicates on the order of one full length-n vector per stage. In contrast, a 2D $\sqrt{p} \times \sqrt{p}$ decomposition avoids assembling a full vector on every rank: each stage matvec is implemented with a column broadcast of a vector panel of length $\approx n/\sqrt{p}$ and a row reduction of partial results of the same size, giving $T_{\text{2D,stage}} \approx 2\alpha \log(\sqrt{p}) + 2\beta(nb/\sqrt{p})$, hence $T_{\text{2D,step}} \approx 4\alpha \log p + 8\beta(nb/\sqrt{p})$. The key takeaway is that 2D reduces the per-rank bandwidth cost by a factor $\sqrt{p}$ relative to 1D (replacing a full-vector allgather by panel broadcast+row reduction), although some layouts may add a modest extra redistribution/transpose term between stages. In summary, the compute cost is the same, but the communication cost is different.

3 Parallel Programming Models

In all implementations we use the same RK4 time stepping scheme. The only difference between models is how the core update kernel `rk4_step_update` is executed in parallel and how the global matrix L and state vector r are partitioned across workers. Figure 1 summarizes the data decomposition for the shared memory, distributed memory, and GPU settings. In the following subsections we show the core parallel kernel for each programming model and briefly describe how work and data are mapped to hardware.

3.1 Shared memory parallelism (OpenMP)

In the OpenMP version the entire matrix and state vector reside in the memory of a single node. Rows of L and the corresponding entries of r are divided among threads, and each thread computes its own block of the RK4 stage vectors. The main parallel region is a loop over row indices annotated with `#pragma omp parallel for` and a vectorized inner loop over columns.

3.2 Single GPU parallelism (CUDA)

The single GPU version keeps the full matrix and state vector on one device. The RK4 stages are implemented as CUDA kernels that assign rows of L to GPU thread blocks. No inter-device communication is required, and only host–device transfers at the beginning and end of the simulation are needed.

3.3 MPI + OpenMP

In the hybrid MPI plus OpenMP version the global row range is partitioned into contiguous blocks and assigned to MPI ranks. Each rank stores its local block of rows of L and the matching segment of r. Before each RK4 stage the ranks exchange their local segments of the current stage input and assemble a full copy of the vector on every rank using a collective communication. Each rank then applies the OpenMP kernel from the shared memory version to its local rows.

Listing 1: MPI+OpenMP RK4 step pseudocode

```
procedure RK4_step_distributed(
    L_local,           // local block of matrix L (rows owned by this rank)
    r_local,           // local part of vector r
    k1_local, k2_local, k3_local, k4_local,
    tmp_local,
    r_global,          // global gather buffer for stage input vector
    D, dt,             // global dimension and time step
    M_loc,             // local number of rows
    recvcounts,        // recv counts for MPI_Allgatherv
    displs,            // displacements for MPI_Allgatherv
    comm               // MPI communicator
)

    // k1 = L r
    MPI_Allgatherv(r_local -> r_global, recvcounts, displs, comm)
    k1_local = L_local * r_global

    // tmp = r + (dt/2) * k1
    tmp_local = r_local
    tmp_local = tmp_local + (dt / 2) * k1_local

    // k2 = L (r + dt/2 k1)
    MPI_Allgatherv(tmp_local -> r_global, recvcounts, displs, comm)
    k2_local = L_local * r_global

    // tmp = r + (dt/2) * k2
```

```
    tmp_local = r_local
    tmp_local = tmp_local + (dt / 2) * k2_local

    // k3 = L (r + dt/2 k2)
    MPI_Allgatherv(tmp_local -> r_global, recvcounts, displs, comm)
    k3_local = L_local * r_global

    // tmp = r + dt * k3
    tmp_local = r_local
    tmp_local = tmp_local + dt * k3_local

    // k4 = L (r + dt k3)
    MPI_Allgatherv(tmp_local -> r_global, recvcounts, displs, comm)
    k4_local = L_local * r_global

    // r = r + dt/6 (k1 + 2 k2 + 2 k3 + k4)
    for i from 0 to M_loc - 1 do
        r_local[i] = r_local[i]
                   + (dt / 6) * (k1_local[i]
                                 + 2 * k2_local[i]
                                 + 2 * k3_local[i]
                                 + k4_local[i])
    end for

end procedure
```

3.4 MPI + CUDA

The MPI plus CUDA implementation uses the same row-wise decomposition as the MPI+OpenMP version, but assigns each block of rows to a different GPU (one MPI rank per GPU). Each rank keeps its local block of L and its local segment of r in device memory, and participates in a collective exchange of the stage input vector before each RK4 stage. The local computation is then performed by a CUDA kernel on the device.

4 Tests and Results

4.1 Tests

To evaluate both numerical accuracy and parallel performance, we carried out a series of experiments on the linear test problem introduced in Section 1. Since we primarily care about the parallel overheads and kernel performance, we use a diagonal L for the accuracy test because it admits an exact solution for validation. Recall that we solve

$$\frac{d\mathbf{r}}{dt} = L\mathbf{r}, \qquad \mathbf{r}(0) = \mathbf{1}, \tag{1}$$

where $L \in \mathbb{R}^{D\times D}$ is diagonal with entries

$$L_{ii} = -1 - 0.1\, i, \qquad i = 0, 1, \ldots, D-1. \tag{2}$$

Table 1: Summary of parallel runs for the linear test problem $\dot{\mathbf{r}} = L\mathbf{r}$ with diagonal $L_{ii} = -1 - 0.1\,i$. For each configuration we vary the global dimension D, advance with RK4 using time step $\Delta t = 10^{-3}$ for n_{steps} steps, and record the maximum wall-clock time over all ranks along with the final ℓ^2 error.

Method	Parameter	Symbol	Values	Notes
MPI+OpenMP	Global size	D	$2^4, 2^5, \ldots, 2^{15}$	Diagonal L stored in dense $D \times D$
	MPI ranks	p	$1, 2, 4, 8, 16$	Distributed over CPU nodes
	OMP threads / rank	t	$1, 2, 4$	`OMP_NUM_THREADS` $= t$
	Time step	Δt	10^{-3}	RK4, fixed across runs
	Number of steps	n_{steps}	20	$T = n_{\text{steps}}\Delta t$
	Output	–	max wall time, ℓ^2 error	
MPI+CUDA	Global size	D	$2^4, 2^5, \ldots, 2^{15}$	Same test problem
	GPUs (MPI ranks)	g	$1, 2, 4$	One MPI rank per GPU
	Time step	Δt	10^{-3}	RK4 on GPUs
	Number of steps	n_{steps}	20	$T = n_{\text{steps}}\Delta t$
	Output	–	max wall time, ℓ^2 error	

For this choice, every component decouples and the exact solution is known in closed form,

$$r_i(t) = \exp\big((-1 - 0.1\,i)\,t\big), \tag{3}$$

which allows us to quantify accuracy a posteriori. At the final time T we assemble the global numerical solution $\mathbf{r}_{\text{num}}(T)$ on rank 0 and compute the global ℓ^2 error

$$\|\mathbf{r}_{\text{num}}(T) - \mathbf{r}_{\text{exact}}(T)\|_2 = \left(\sum_{i=0}^{D-1} \big(r_{\text{num},i}(T) - \exp((-1 - 0.1\,i)\,T)\big)^2\right)^{1/2}. \tag{4}$$

All experiments use the same fourth-order Runge–Kutta time integrator with time step $\Delta t = 10^{-3}$ and a fixed number of steps $n_{\text{steps}} = 20$, so that $T = n_{\text{steps}}\Delta t$.

4.1.1 Sweep Design and Parallel Configurations

We conducted a systematic parameter sweep over both problem size and parallel configuration. The global dimension D was varied as powers of two,

$$D \in \{2^4, 2^5, \ldots, 2^{15}\},$$

in order to explore both small and moderately large matrices while keeping the eigenvalue spectrum and RK4 stability constraints under control. For each $(D, \Delta t, n_{\text{steps}})$ configuration we recorded (i) the maximum wall-clock time over all ranks and (ii) the final ℓ^2 error defined above. Wall-clock time is measured using `MPI_Wtime`; we report the maximum over ranks so that the timing reflects the true parallel runtime.

On the CPU, we benchmark a hybrid MPI+OpenMP implementation. We vary the number of MPI ranks

$$p \in \{1, 2, 4, 8, 16\}$$

and, independently, the number of OpenMP threads per rank,

$$t \in \{1, 2, 4\},$$

subject to node core limits. For each triple (D, p, t) we run the full time integration and write a CSV file containing D, T, Δt, n_{steps}, the measured wall time, and the ℓ^2 error. These data allow us to extract (i) strong-scaling curves by fixing D and t and varying p, (ii) thread scaling by fixing D and p and varying t, and (iii) scaling trends under increasing global problem size.

On the GPU, we benchmark a hybrid MPI+CUDA implementation in which one MPI rank controls one GPU and holds a disjoint block of rows of L. We vary the number of GPUs

$$g \in \{1, 2, 4\},$$

again sweeping over the same set of problem sizes $D = 2^4, \ldots, 2^{15}$. As in the CPU experiments, each (D, g) run produces a timing and error record. From this sweep we form GPU strong-scaling plots by fixing D and varying g.

Note on "weak scaling" for dense matvec. For a dense matrix–vector product with a 1D row-wise decomposition, the local arithmetic work per rank scales like $\Theta(D^2/p)$. True weak scaling (constant local work) therefore corresponds to choosing $D \propto \sqrt{p}$. In our experiments, we additionally report a commonly used scaling study in which the number of owned rows per rank is held fixed (i.e., D/p is approximately constant, so $D \propto p$). For dense operators, this fixed-row study does *not* hold local work constant because the inner dimension of each dot product still grows with D; consequently, runtime growth in this study reflects both increased arithmetic cost and communication costs (such as the per-stage `MPI_Allgatherv`).

4.2 Results

4.2.1 Accuracy

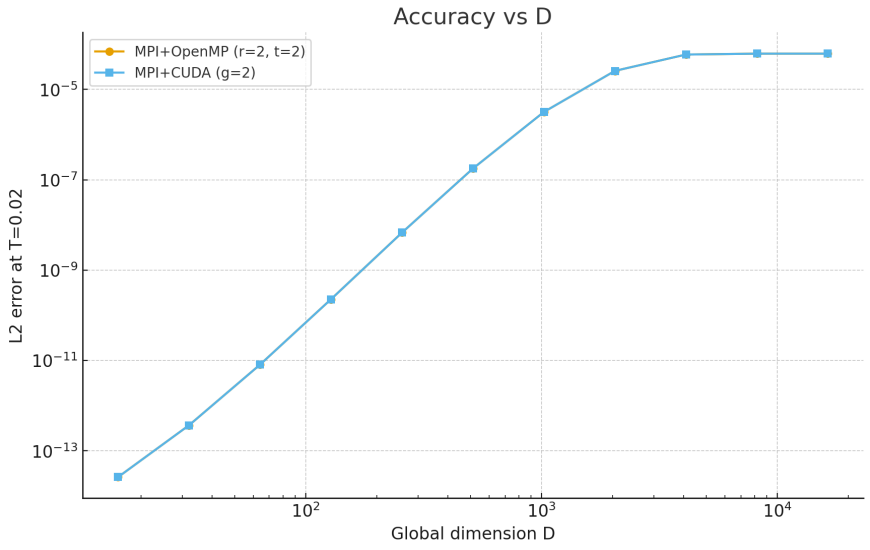

Figure 2: Accuracy (error) vs. problem size D for fixed $\Delta t = 10^{-3}$ and $T = 0.02$.

Table 2: Accuracy of the RK4 integrator for the diagonal test problem $\dot{\mathbf{r}} = L\mathbf{r}$ with $L_{ii} = -1 - 0.1\,i$. We report the ℓ^2 error at the final time $T = 0.02$ between the numerical solution and the exact solution $r_i(T) = \exp((-1 - 0.1\,i)T)$. GPU runs use the MPI+CUDA implementation with two GPUs ($g = 2$); CPU runs use MPI+OpenMP with two MPI ranks and two OpenMP threads per rank ($p = 2$, $t = 2$).

D	ℓ^2 error (MPI+CUDA, $g = 2$)	ℓ^2 error (MPI+OpenMP, $p = 2$, $t = 2$)
16	2.63×10^{-14}	2.63×10^{-14}
32	3.70×10^{-13}	3.70×10^{-13}
64	8.12×10^{-12}	8.12×10^{-12}
128	2.28×10^{-10}	2.28×10^{-10}
256	6.79×10^{-9}	6.79×10^{-9}
512	1.78×10^{-7}	1.78×10^{-7}
1024	3.19×10^{-6}	3.19×10^{-6}
2048	2.58×10^{-5}	2.58×10^{-5}
4096	5.93×10^{-5}	5.93×10^{-5}
8192	6.20×10^{-5}	6.20×10^{-5}
16384	6.20×10^{-5}	6.20×10^{-5}
32768	1.99×10^{7}	1.99×10^{7}

4.2.2 Performance vs. D

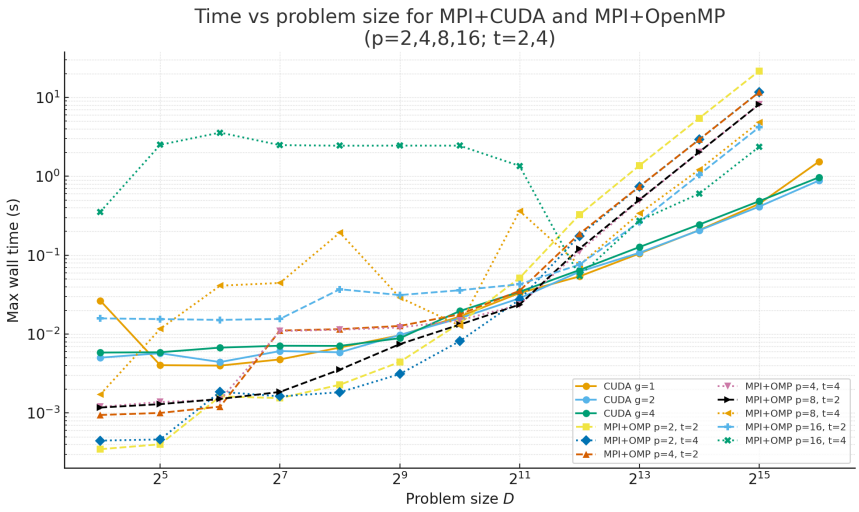

Figure 3: Wall-clock time vs. problem size D for different settings.

4.2.3 Strong scaling

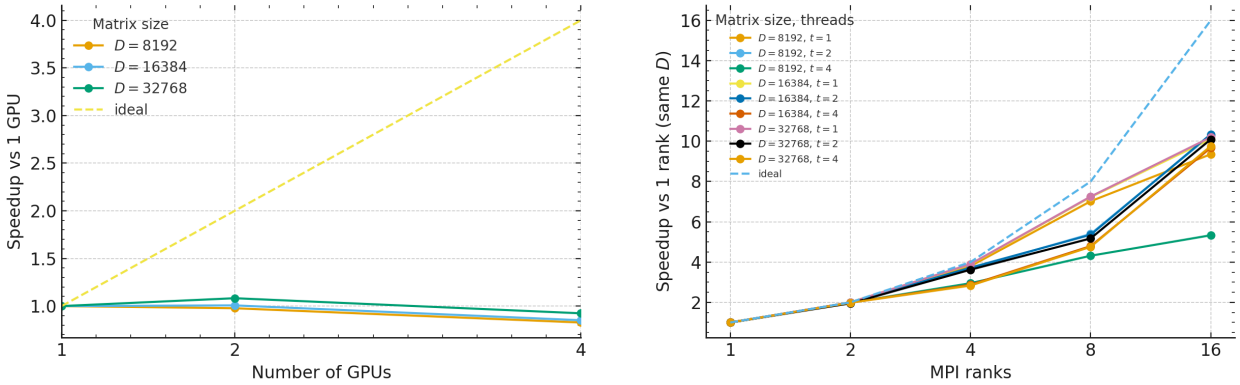

Figure 4: Strong scaling studies for MPI+CUDA (left) and MPI+OpenMP (right).

4.2.4 Weak scaling

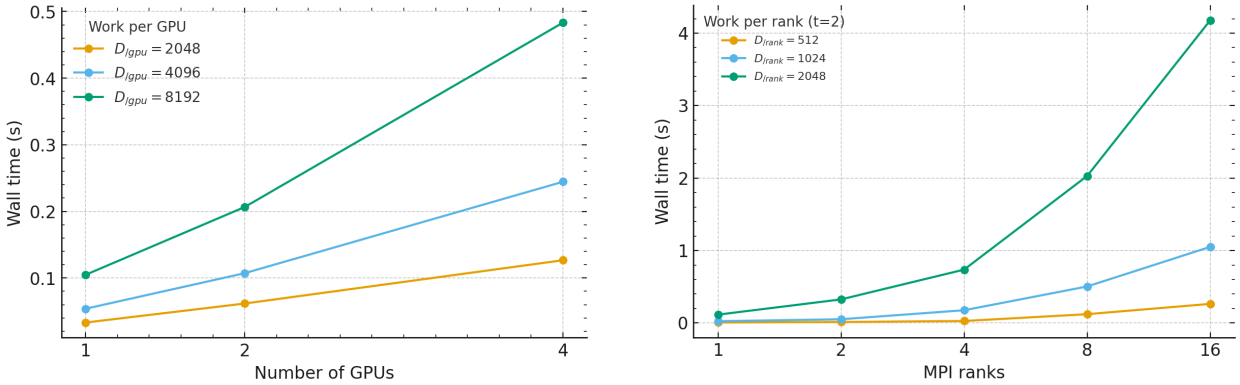

Figure 5: Scaling study under increasing global problem size for MPI+CUDA (left) and MPI+OpenMP (right).

5 Discussion

5.1 Accuracy and Numerical Consistency

Table 2 reports representative results for the MPI+CUDA implementation using two GPUs and the MPI+OpenMP implementation using two MPI ranks and two OpenMP threads per rank. For all problem sizes in the range $16 \leq D \leq 16384$, the two implementations produce errors that agree to all printed digits. For very small dimensions ($D = 16$), the error is at the level of machine round-off ($\mathcal{O}(10^{-14})$), while for larger D the error saturates around $\mathcal{O}(10^{-5}$–$10^{-4})$ at fixed $\Delta t = 10^{-3}$. (A separate Δt refinement test, not shown here, verifies fourth-order convergence of the RK4 time integrator in the stable regime.)

An important feature of the data is the consistency across parallel configurations. For MPI+OpenMP, the ℓ^2 error is essentially independent of the number of MPI ranks p and threads t used, as long as the configuration remains within the stable regime. This behavior is expected, since the hybrid

MPI+OpenMP algorithm is algebraically identical to the serial RK4 update: MPI partitions rows of L, and OpenMP parallelizes independent loops, but no additional numerical approximations are introduced. Similarly, for MPI+CUDA, the ℓ^2 errors obtained with $g = 1$, $g = 2$, and $g = 4$ GPUs are indistinguishable for all $D \leq 16384$. This confirms that distributing the matrix–vector operations across multiple accelerators does not alter the numerical solution, and that all GPU kernels faithfully reproduce the intended RK4 update.

A sharp change occurs once D reaches $2^{15} = 32768$. For $D = 32768$ and above, both the MPI+OpenMP and MPI+CUDA results become numerically unstable, with the ℓ^2 error growing catastrophically. This behavior is explained by the stability restrictions of explicit Runge–Kutta methods. The eigenvalues of L are $\lambda_i = -1 - 0.1\,i$, so the most negative eigenvalue grows linearly with D:

$$\lambda_{\min} \approx -1 - 0.1\,(D - 1).$$

RK4 is stable for scalar problems $\dot{y} = \lambda y$ only when the scaled quantity $z = \lambda\,\Delta t$ lies within its classical stability region, which intersects the negative real axis at approximately $z \approx -2.8$. For $\Delta t = 10^{-3}$, this implies a stability threshold of roughly $|\lambda_{\min}| \lesssim 2800$. At $D = 16384$, we have $|\lambda_{\min}| \approx 1640$, comfortably within the stable regime. At $D = 32768$, however, $|\lambda_{\min}| \approx 3280$, so that $|\lambda_{\min}\Delta t| \approx 3.28$, which lies outside the RK4 stability region. In this regime explicit RK methods, including RK4, exhibit exponential amplification of errors, and the observed loss of accuracy is consistent with this theoretical prediction.

In summary, both MPI+OpenMP and MPI+CUDA produce numerically identical solutions in the stable regime and maintain accuracy independent of the parallel configuration. Instability at large D is not a defect of either implementation but rather a consequence of applying a fixed-step explicit RK method beyond its stability limits.

5.2 Scaling behaviour

We first examine strong scaling by fixing the global problem size and varying the parallel resources. For the CPU implementation, we consider the MPI+OpenMP code with $D = 16384$ and $T = 0.02$. At fixed thread count $t = 2$, increasing the number of MPI ranks from $p = 1$ to $p = 2, 4, 8, 16$ reduces the wall time from 10.9 s to 5.45, 2.92, 2.03, and 1.05 s, respectively. This corresponds to speedups of 2.0×, 3.7×, 5.4×, and 10.4×, with parallel efficiencies remaining above 90% up to $p = 4$ and around 65% at $p = 16$.

In contrast, the MPI+CUDA solver shows limited strong scaling across 1–4 GPUs for the problem sizes tested. For example, at $D = 16384$ and $T = 0.02$ the runtime changes only from 0.208 s on one GPU to 0.207 s and 0.244 s on two and four GPUs, respectively. This behaviour is consistent with the algorithmic structure: each Runge–Kutta stage performs an `MPI_Allgatherv` of the full stage input vector, and any additional overheads from synchronization and host–device data movement can dominate once the local matrix–vector work per GPU becomes sufficiently small.

The "weak scaling" study shown in Figure 5 uses a fixed-row-per-rank design (e.g., holding $D_{\text{local}} = D/p$ approximately constant), so the global dimension increases proportionally with p (or with g for GPUs). For dense matrix–vector products, this implies that local arithmetic cost increases with the number of ranks because each local dot product length grows with D. As a result, runtime growth in this study reflects both increased compute and the increasing cost of global collectives such as `MPI_Allgatherv`. For example, holding $D_{\text{local}} = 4096$ per GPU (i.e., $D = 4096, 8192, 16384$ for $g = 1, 2, 4$ GPUs) leads to runtimes of 0.054, 0.107, and 0.244 s, which is consistent with the increased local work and additional communication.

Finally, we study thread-to-thread scaling within the MPI+OpenMP implementation by fixing D and p and varying the number of OpenMP threads t. For $D = 16384$ and $p = 1$, the runtime decreases from 21.0 s at $t = 1$ to 10.9 s at $t = 2$ and 5.82 s at $t = 4$, corresponding to speedups of 1.9× and 3.6× (efficiencies of 96% and 90%). Similar near-ideal thread scaling is observed for $p = 2$, while at larger p the speedups flatten (e.g., from 2.90 s to 2.03 s to 1.21 s for $p = 8$ and $t = 1, 2, 4$), reflecting increased contention for memory bandwidth and the growing cost of MPI communication. Overall, the CPU code exhibits good strong and thread scaling, whereas multi-node scaling is constrained by the collective communication pattern inherent in the current RK4/MPI design.

5.3 MPI communication size limit

In our experiments we cap the problem size at $D = 2^{15} = 32{,}768$ rather than extending the sweep to $D = 2^{16}$. The reason is that we store the operator $L \in \mathbb{R}^{D\times D}$ as a dense matrix and distribute it with `MPI_Scatterv`, whose `count` arguments are 32-bit integers. With two MPI ranks, each rank owns roughly $(D/2) \cdot D$ entries of L; for $D = 2^{15}$ this is $2^{29} = 536{,}870{,}912$ elements, which is well below `INT_MAX` = 2,147,483,647, but for $D = 2^{16}$ the per-rank block size becomes $2^{31} = 2{,}147{,}483{,}648$ elements and can no longer be represented in a 32-bit `int`. Beyond this point `MPI_Scatterv` would require counts larger than `INT_MAX`, leading to invalid `MPI_ERR_COUNT` failures and, even ignoring MPI, per-rank memory footprints on the order of tens of GiB for L alone. A possible solution, left for future work, would be to implement a chunked distribution scheme that splits each rank's block into multiple `MPI_Scatterv` or `MPI_Send`/`MPI_Recv` calls with `count` $\leq$ `INT_MAX`.

5.4 Load Balancing

The workload is almost evenly distributed across workers because each worker owns an equal block of rows with a difference of one row at most. Load imbalance only becomes noticeable when the number of workers is comparable to or larger than the problem dimension D; otherwise, for sufficiently large D, the imbalance is negligible.

5.5 Memory Usage

Memory usage is a significant limitation for dense formulations. In the time-independent case $L(t) = L$, storing the dense operator dominates memory: a double-precision real dense matrix requires $8D^2$ bytes, while a double-precision complex dense matrix requires $16D^2$ bytes. For example, a single double-precision complex matrix of size 16,384 × 16,384 requires

$$16 \cdot 16384^2 \text{ bytes} \approx 4 \text{ GiB}.$$

The RK4 work vectors (r, k_1–k_4, and a temporary buffer) contribute only $\mathcal{O}(D)$ additional storage.

If the operator is time-dependent and is pre-tabulated as a sequence of matrices $\{L(t_n)\}_{n=1}^{N_t}$, then storing all time slices requires memory proportional to $O(D^2 N_t)$, which can quickly become prohibitive. For instance, storing 100 double-precision complex matrices at $D = 16384$ would require roughly 400 GiB in addition to the work vectors. In that case, a time-local strategy would be needed, where each $L(t)$ is loaded from disk or generated on demand rather than keeping all time slices resident in memory.

In a shared-memory environment (OpenMP on a single node), all threads access the same global arrays, so the memory footprint is dominated by the full $D \times D$ matrix L, plus $\mathcal{O}(D)$ work vectors. If the full trajectory is stored for n_t time steps, an additional $\Theta(Dn_t)$ storage is required. In a

distributed-memory environment with p MPI ranks, each rank stores only its local block of L of size $(D/p) \times D$, reducing the per-rank matrix cost to $O(D^2/p)$, and similarly holds only D/p entries of each work vector. The only fully replicated array in the current 1D row-wise algorithm is the gathered stage input vector of length D, adding $O(D)$ bytes per rank. Although the global memory across all ranks still sums to $O(D^2)$ for the full matrix, the peak per-node memory is reduced by a factor of p, which often makes larger D feasible.

6 Conclusion

In this project, we evaluated a parallel Runge–Kutta 4 (RK4) integrator implemented with two hybrid programming models, MPI+CUDA and MPI+OpenMP, and also developed single-node baselines (OpenMP and single-GPU CUDA). Our design employs row-wise data parallelism, distributing disjoint blocks of the system matrix L to workers while requiring a global `MPI_Allgatherv` at every RK4 stage so that each rank or GPU has access to the full stage input vector needed for its local matrix–vector multiplication. Numerically, the method achieves the expected RK4 behavior in the stable regime, and solutions are consistent across programming models until the problem size becomes large enough to push the fixed-step explicit scheme outside its stability region, at which point the solution rapidly deteriorates.

Performance studies reveal that global collective communication and synchronization overheads are the dominant bottlenecks in multi-node and multi-GPU executions, limiting scalability once the per-rank matrix–vector work becomes small. Memory growth is dominated by storing the dense matrix L, which scales as $O(D^2)$ for time-independent operators; for time-dependent operators tabulated over time, the storage can scale as $O(D^2 N_t)$. Large problem sizes also approach practical MPI limits (e.g., 32-bit count arguments in `MPI_Scatterv`), motivating alternative distribution schemes or communication-avoiding formulations. Future work should therefore explore (i) 2D matrix decompositions to reduce per-rank communication volume, and (ii) reformulations that reduce or eliminate global vector assembly at each RK4 stage.

References

[1] John C. Butcher. *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons, 2 edition, 2008.

[2] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge University Press, 2 edition, 2009.

[3] Heinz-Peter Breuer and Francesco Petruccione. *The Theory of Open Quantum Systems*. Oxford University Press, 2002.

[4] Ángel Rivas and Susana F. Huelga. *Open Quantum Systems: An Introduction*. Springer, 2012.

[5] Göran Lindblad. On the generators of quantum dynamical semigroups. *Communications in Mathematical Physics*, 48:119–130, 1976.

[6] Vittorio Gorini, Andrzej Kossakowski, and E. C. G. Sudarshan. Completely positive dynamical semigroups of n-level systems. *Journal of Mathematical Physics*, 17(5):821–825, 1976.

[7] Erwin Fehlberg. Low-order classical runge-kutta formulas with stepsize control and their application to some heat transfer problems. *NASA Technical Report R-315*, 1969.

[8] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

[9] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010.

[10] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Programming with the Message Passing Interface*. MIT Press, 1999.

[11] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.