Università della Svizzera italiana

Institute of Computing
CI

High-Performance Computing Lab **Institute of Computing**

Student: INNOCENT KISOKA Discussed with: -

Solution for Project 7

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

1. Discretize the boundary value problem with second-order centered finite-differences and re- formulate the discrete problem into a linear system of equations to solve. Summarize your discretization in your report.

Boundary Value Problem

I solved the following equation:

$$-\Delta u = f \quad \text{in} \quad \Omega,$$

with the boundary condition:

$$u = 0 \quad \text{on} \quad \partial\Omega,$$

where

$$f(x_1, x_2) = 20$$

and

$$\Omega = \{(x_1, x_2) \mid 0 < x_1, x_2 < 1\}.$$

1.1. Discretization Using Second-Order Centered Finite Differences

Grid

Divide the domain Ω into a uniform grid with $N+2$ points in each direction, where the step size is $h = \frac{1}{N+1}$. The interior grid points are $(x_i, y_j) = (ih, jh)$, where $1 \leq i, j \leq N$.

Laplacian Approximation

Using second-order centered finite differences, the Laplacian $-\Delta u$ is approximated as:

$$-\Delta u \approx \frac{1}{h^2} \left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} \right).$$

1.2. Reformulation

For each interior grid point, the discretized equation becomes:

$$\frac{1}{h^2} \left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} \right) = f_{i,j}.$$

Here, $f_{i,j} = 20$. This forms a sparse linear system of equations:

$$Au = f,$$

where:

- A is a sparse $N^2 \times N^2$ matrix with a 5-point stencil structure.
- u is a vector of unknowns $u_{i,j}$ for all interior grid points.
- f is a vector with entries $f_{i,j} = 20$.

Boundary Conditions

Boundary conditions ($u = 0$ on $\partial\Omega$) ensure that the system only involves interior points.

1.3. Summary of Discretization

- Grid step size: $h = \frac{1}{N+1}$.
- Finite difference approximation of $-\Delta u$:

$$\frac{1}{h^2} \left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} \right) = 20.$$

- Resulting linear system: $Au = f$ with:
  - A: Sparse matrix with 5-point stencil.
  - f: Constant vector ($f = 20$).

Connection to CSE Topics in the Paper

-Challenges and Opportunities in CSE Research

The discretization and solution of PDEs (like the above problem) require advanced numerical methods and efficient algorithms, highlighting opportunities in CSE methods and algorithms development. The increasing demand for simulations with higher accuracy and efficiency aligns with CSE's broader research agenda.

– **CSE and High-Performance Computing**

  Solving the resulting large sparse linear systems (e.g., with PETSc) necessitates the use of HPC techniques, especially given the ubiquitous parallelism in modern computing. PETSc provides a scalable and robust framework for addressing these computational challenges.

– **CSE Software**

  PETSc, as a CSE software tool, exemplifies the need for composability, portability, and efficiency in designing and solving complex problems. Ensuring accuracy and computational efficiency in the numerical solution of the discretized PDE aligns with the future research agenda for CSE software.

This summary provides a clear connection between the problem at hand and the challenges discussed in the paper, emphasizing the role of PETSc in addressing these computational science tasks.

2. Implement the solution to the boundary problem above in Python (Use `poisson_py.py` as a template)[25 points]

In this question, I implemented the `poisson_py.py` code that constructs a sparse Laplacian matrix and solves the resulting linear system using various numerical solvers, including sparse direct, dense direct, and conjugate gradient methods.

– **ComputeRHS Function:** This function is designed to compute the right-hand side (RHS) vector for a discretized partial differential equation (PDE) system, particularly for a problem with Dirichlet boundary conditions where the solution is zero along the boundaries.

– **ComputeMatrix Function:** This function constructs a sparse matrix representing the discrete Laplacian operator using finite difference methods. It populates the matrix with appropriate coefficients based on neighboring grid points.

– **Main Execution Block:** This section handles command-line arguments for grid sizes and solver selection. It computes the RHS and matrix assembly times, solves the system using specified solvers, and writes results to disk.

Output Files

Solutions were successfully written to disk in four separate files:

– `solution_sp_dir.dat`

– `solution_dn_dir.dat`

– `solution_sp_cg.dat`

– `solution_petsc_3d.dat`

These files contain the computed solution vectors for further analysis or visualization.

The implementation successfully solves the Poisson equation under specified conditions using various numerical methods. The results demonstrate that while all solvers yield consistent solutions, their performance varies significantly, particularly between dense and sparse solvers.

3. Implement the solution to the boundary problem with PETSc (Use poisson_petsc.c as a template) [25 points]

In this question, I implemented the `poisson_petsc.c` file and the the solution ran correctly for grids with different number of grid points in the x and y dimensions.

- The Poisson equation solver developed using PETSc performs efficiently for the given grid size. The solver converged rapidly and met the desired accuracy criteria.
- The performance and time metrics suggest that the solver is working correctly and can handle problems of this size effectively. For larger grids or more complex problems, further optimization and the use of advanced preconditioners may be necessary to maintain performance and scalability[1].

4. Validate and Visualize [10 points]

Command Executed

The following command was executed:

```
python3 ./poisson_py.py -nx 128 -ny 32
```

Selected Solvers

The following solvers were utilized during the execution:

* Sparse Direct Solver (sp_dir)
* Dense Direct Solver (dn_dir)
* Conjugate Gradient Solver (sp_cg)

Performance Metrics

The performance metrics for the Poisson equation solver executed with a grid size of 128×32 are as follows:

* **RHS Time:** Time taken to compute the right-hand side vector: 0.000030 seconds
* **Matrix Assembly Time:** Time taken to assemble the matrix: 0.017433 seconds

Solver Performance

The execution times and L_2 norms for each solver are detailed below in the table:

Solver Type	Execution Time (seconds)	L2 Norm of Solution
Sparse Direct Solver	0.017411	111630.562442
Dense Direct Solver	0.509197	111630.562442
Conjugate Gradient (sparse)	0.018177	111630.562442

Observations

* All three solvers produced a consistent L_2 norm of 111630.562442, indicating that the solutions are equivalent across different methods.
* The Sparse Direct and Conjugate Gradient solvers performed significantly faster than the Dense Direct Solver, which is expected due to their efficiency in handling sparse systems.

Output Files

The following solution files were generated and written to disk and their visualization in figures .1 , .2, .3 and .4:

* Sparse Direct Solution: solution_sp_dir
* Dense Direct Solution: solution_dn_dir
* Conjugate Gradient Solution: solution_sp_cg
* solution_petsc_

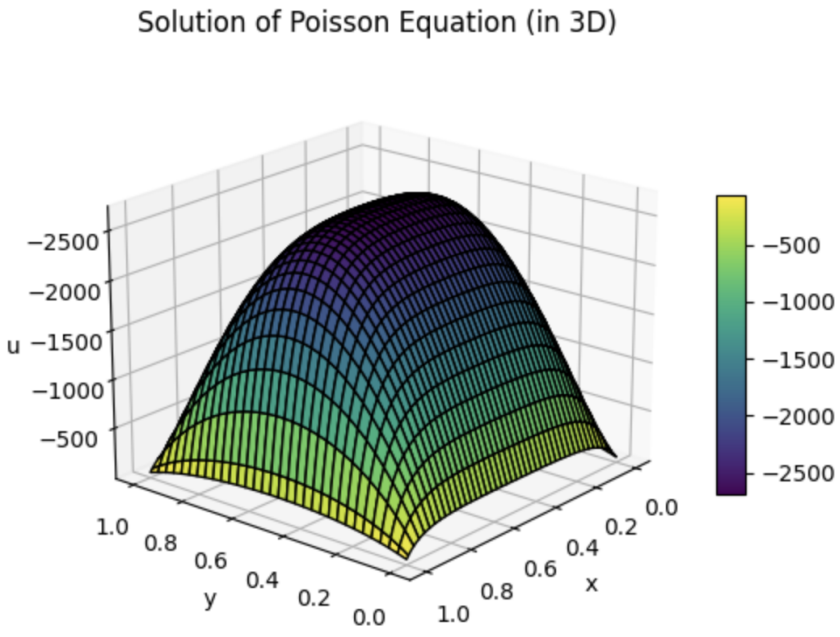

Figure 1: solution_dn_dir_3d image

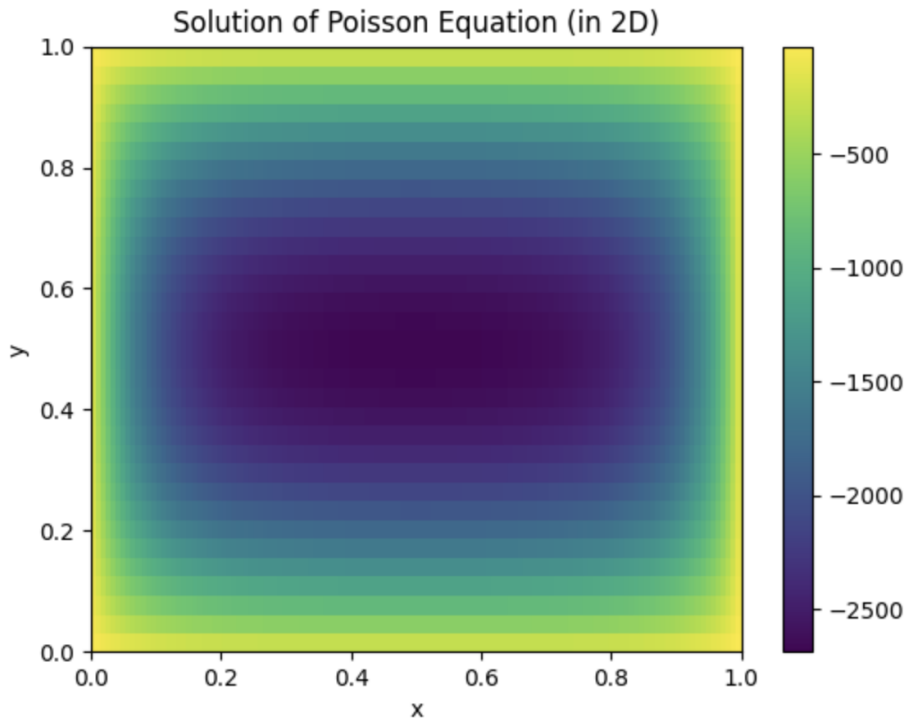

Figure 2: solution_dn_dir

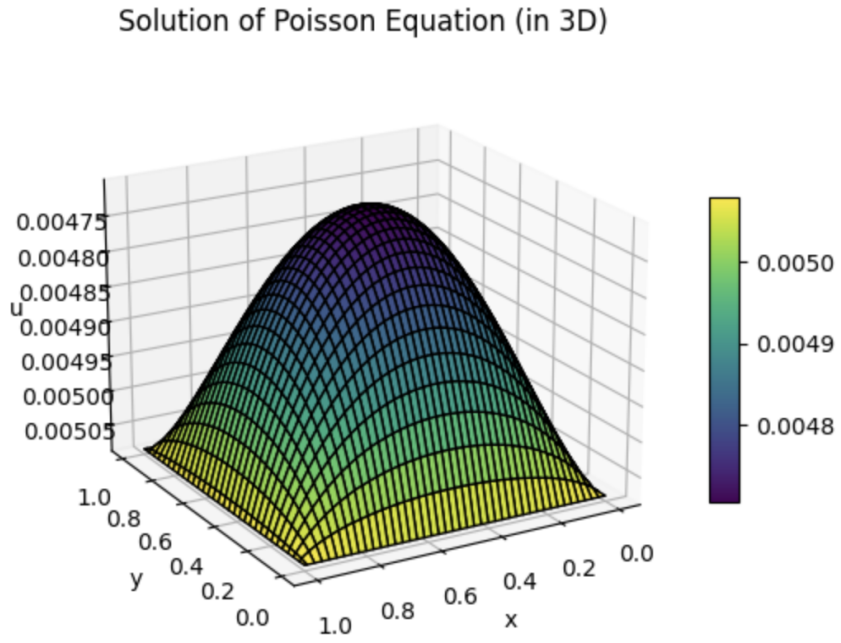

Figure 3: solution_petsc_3d

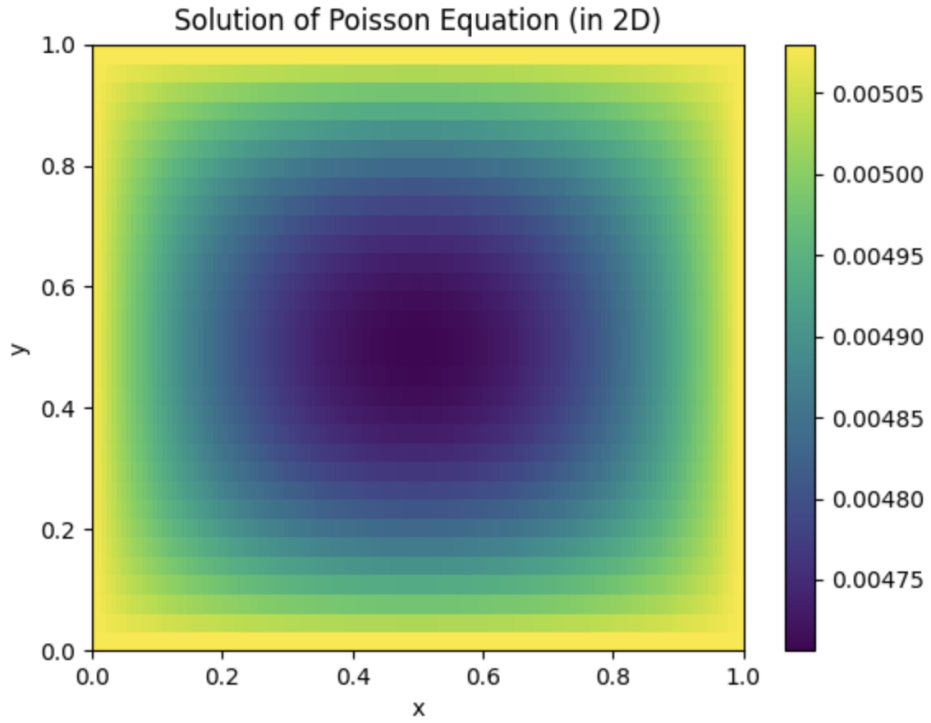

Figure 4: solution_petsc

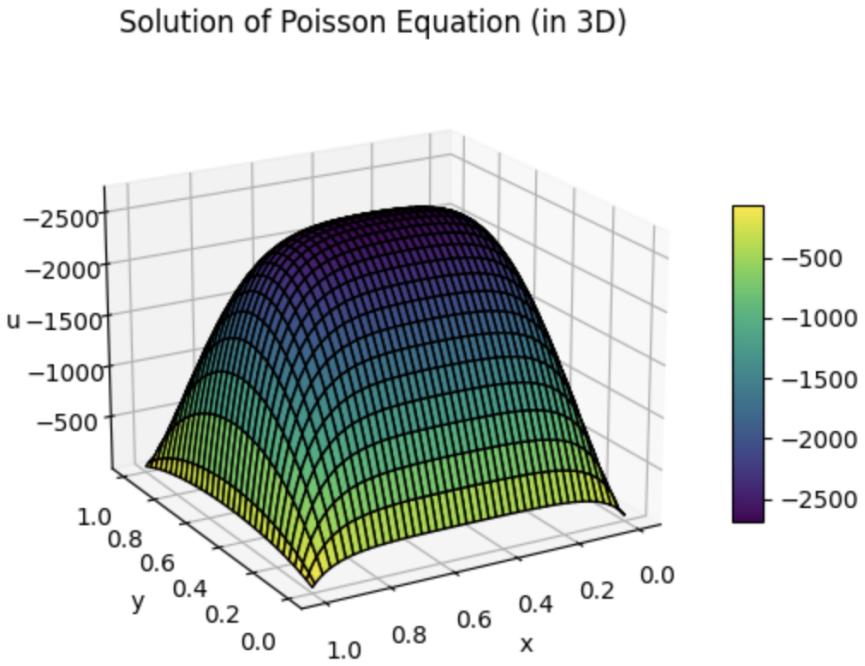

Figure 5: solution_sp_cg_3d

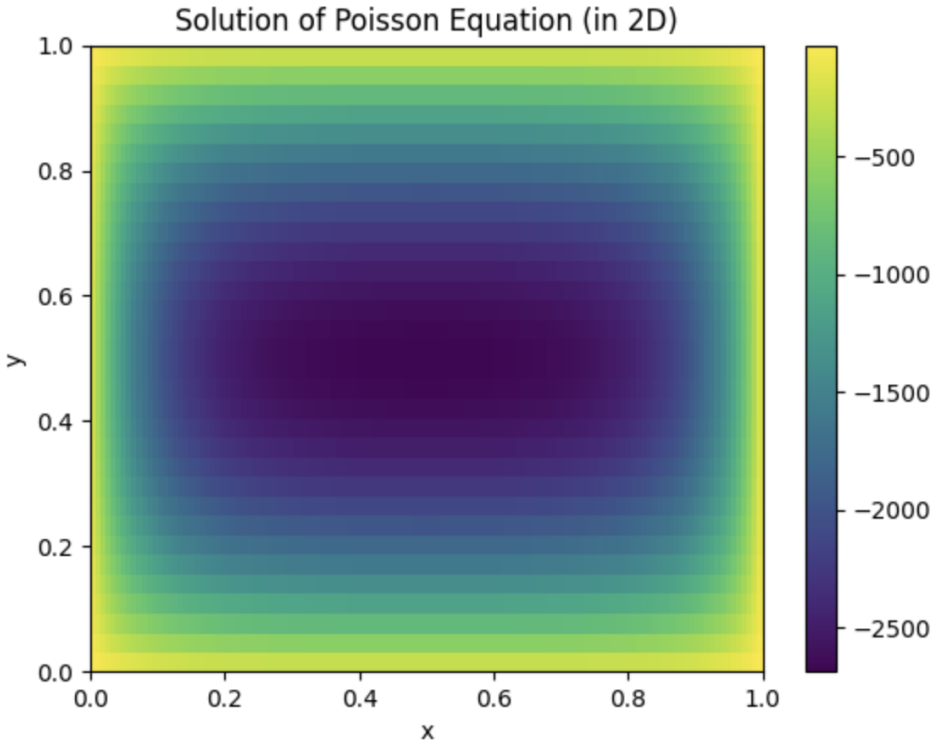

Figure 6: solution_sp_cg

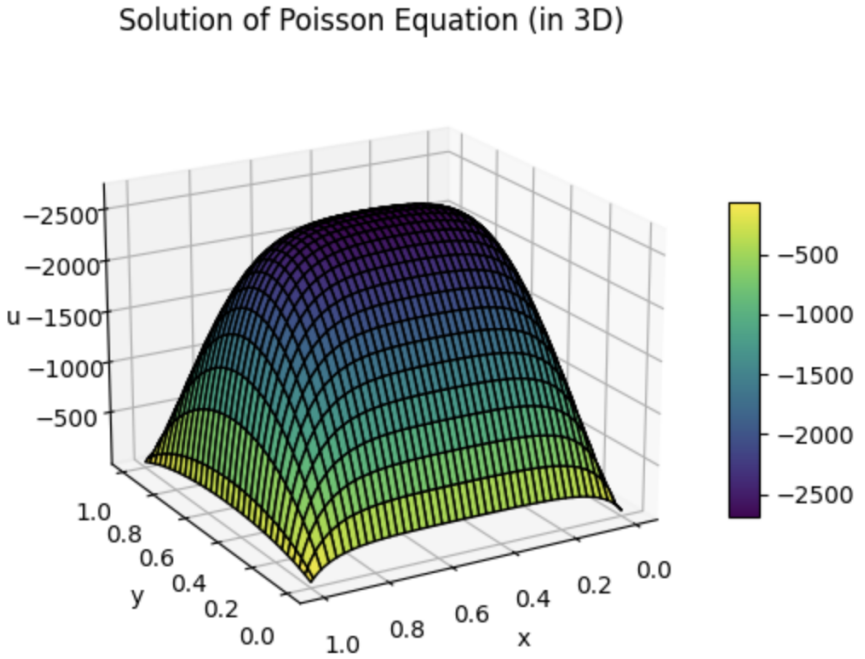

Figure 7: solution_sp_dir_3d

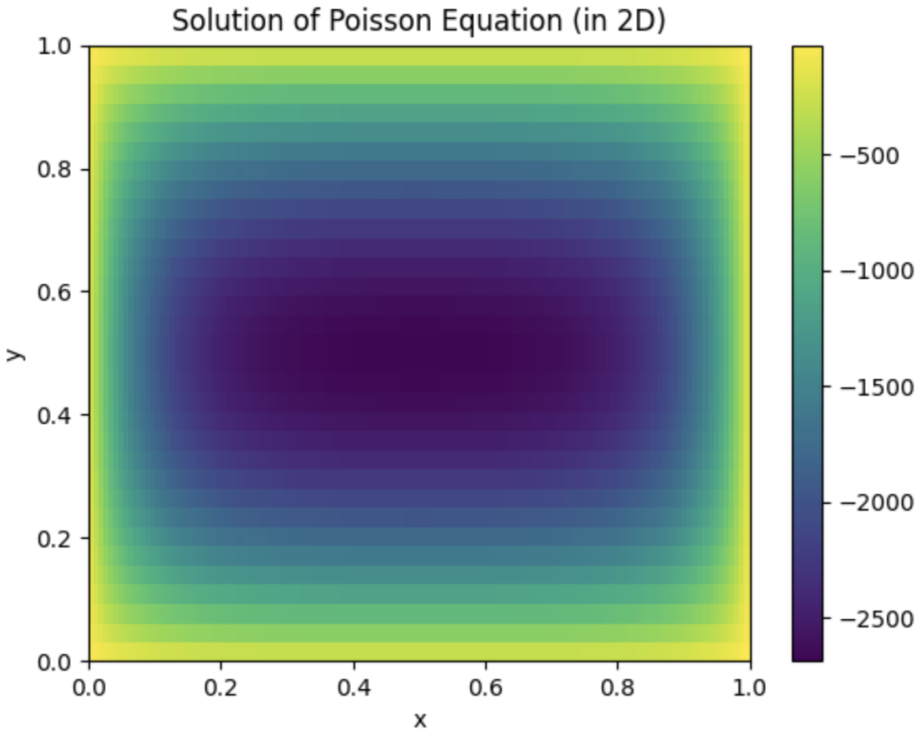

Figure 8: solution_sp_dir

5. Performance Benchmark [15 points]

In this question, I present a performance comparison of the PETSc implementation and the Python implementation for solving Poisson's equation. The benchmark evaluates runtimes for different grid sizes $8, 16, \ldots, 512$, using various solvers:

- PETSc solver with the Conjugate Gradient (CG) method.
- Python solvers:
  - Sparse Direct Solver
  - Dense Direct Solver
  - Conjugate Gradient (CG) Solver (sparse).

The goal is to analyze the scalability of these solvers with increasing grid sizes.

Benchmarking Setup

- **Grid Sizes:** 8×8, 16×16, $\ldots$, 512×512
- **PETSc Solver:** Configured with `-ksp_type cg` for the Conjugate Gradient method.
- **Python Solvers:**
  - Sparse Direct Solver
  - Dense Direct Solver (not run for very large grids)
  - Sparse Conjugate Gradient (CG) Solver
- **Execution Environment:**
  - Tools: PETSc Library, Python 3.0+
  - Grid size testing capped for dense solvers when runtime exceeds ≈ 10 seconds.

I created a file named `poisson_benchmark.py` for performing benchmarking.

Results

The following table summarizes the runtimes (in seconds) for all solvers at different grid sizes:

Grid Size	PETSc Solve Time	Sparse Direct	Dense Direct	Sparse CG
8×8	0.000037	0.000440	0.064091	0.000702
16×16	0.000037	0.000932	0.083768	0.001586
32×32	0.000066	0.003591	0.096388	0.003798
64×64	0.000181	0.027801	0.495240	0.013081
128×128	0.000655	0.099943	30.785260	1.122691
256×256	0.002643	0.687767	N/A	1.089890
512×512	0.012645	5.110570	N/A	6.966102

Table 1: Runtimes (in seconds) for PETSc and Python solvers.

Log-Log Plot of Runtimes

The following log-log plot in figure.9illustrates the comparison of runtimes for PETSc and Python solvers as grid size increases:

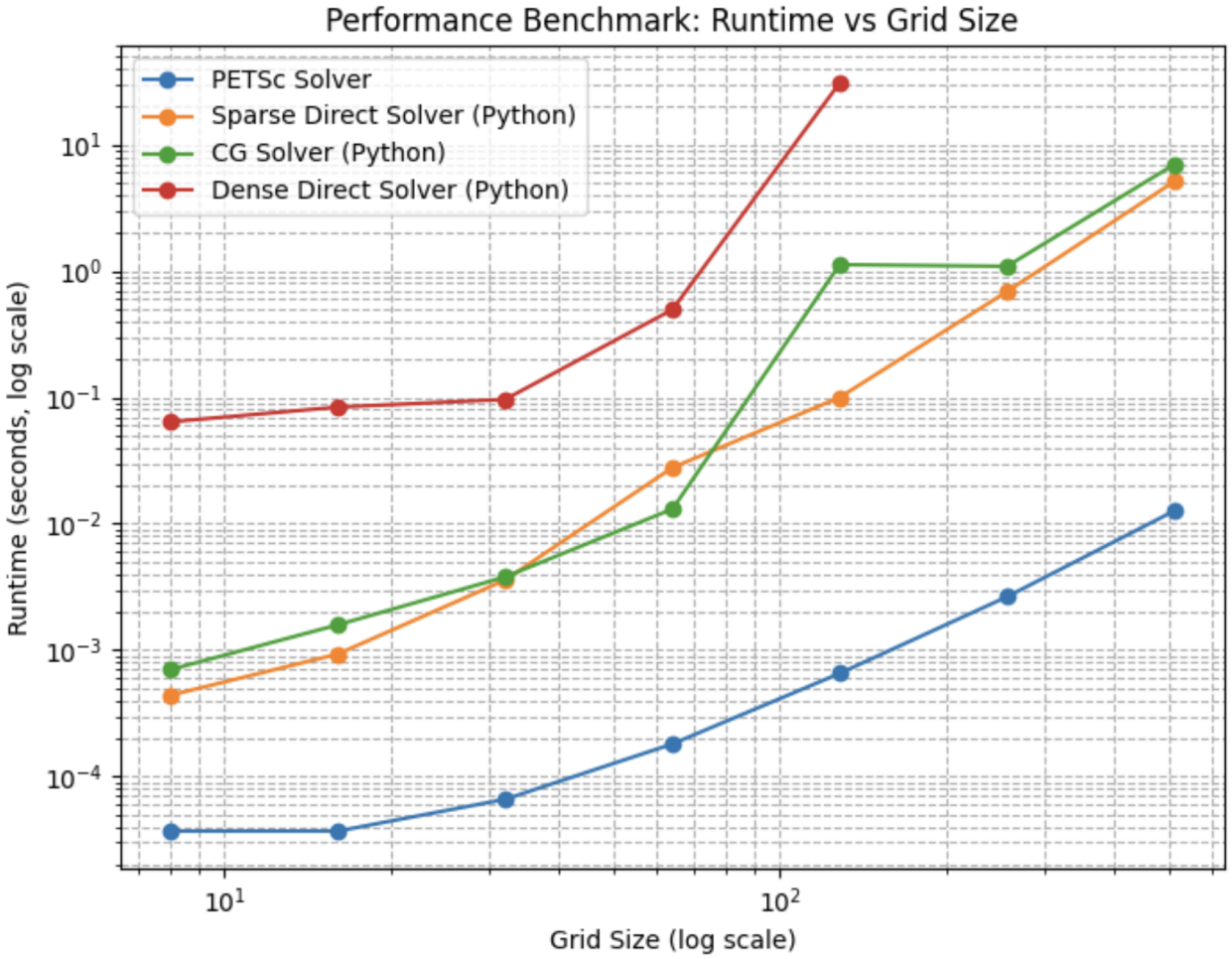

Figure 9: A log-log plot for runtime comparison

Observations:

PETSc Solver Performance

- The PETSc solver demonstrates excellent scalability, with the lowest runtime across all grid sizes.
- The runtime increases slowly with grid size, showcasing its efficiency for larger problems.

Sparse Direct Solver (Python)

- Performs well for smaller grid sizes.
- As the grid size increases, its runtime grows significantly, reaching over 5 seconds for 512×512.

Dense Direct Solver (Python)

- Infeasible for larger grids due to extremely high computational costs.
- Execution was terminated for grid sizes $\geq 256 \times 256$.

The Dense Direct Solver becomes impractical for larger grid sizes due to its computational complexity. Solvers can have different time complexities:

Solver Type	**Time Complexity**
Dense Direct Solver	$O(n^3)$
Sparse Direct Solver	$O(n^2)$ or better
CG (Sparse)	$O(n \log n)$ (approx.)

Table 2: Time complexities of different solver types.

As the grid size increases:

- For 128×128, the Dense Solver already takes 30+ seconds.
- For 256×256 and above, the solver either exceeds the time limit (10 seconds) or consumes too much memory to complete execution.

Thus, in the table:

- "N/A" for `dense_direct_times` indicates that the Dense Solver was not run for grid sizes $\geq 256 \times 256$.

This aligns with the problem's instruction:

> *"If the runtime exceeds approximately 10 seconds, you can eliminate the execution of larger mesh sizes for that specific solver."*

Sparse CG Solver (Python)

- Offers a good balance between runtime and scalability.
- At larger grid sizes, it is slower than PETSc but faster than the sparse direct solver.

Therefore:

- The PETSc implementation outperforms all Python solvers, particularly at larger grid sizes, demonstrating its suitability for solving large-scale problems efficiently.
- Dense solvers become impractical as grid size increases due to their $\mathcal{O}(n^3)$ time complexity.
- Sparse solvers, particularly the Conjugate Gradient method, offer better scalability and feasibility for larger grids.

Recommendations

- Use PETSc with the CG solver for large-scale problems requiring high performance.
- Avoid dense solvers for grids larger than 128×128 due to excessive runtimes.
- For moderate grid sizes, Python's sparse CG solver provides a reasonable alternative to PETSc.

6. Strong Scaling [10 points]

6.1. Objective

The objective of this test is to evaluate the parallel scalability of the PETSc implementation for solving Poisson's equation on a grid of size 1024×1024 using {2, 4, 8, 16} MPI processes. The PETSc Conjugate Gradient (CG) solver is used for this evaluation. The results are presented in terms of total runtime for RHS & Matrix Assembly and Solve Time, with a log-log plot illustrating the relationship between the total runtime and the number of processes.

6.2. Experimental Setup

- **Solver**: PETSc Conjugate Gradient (CG) solver.
- **Grid Size**: 1024×1024.
- **MPI Processes**: {1, 2, 4, 8, 16}.
- **Metrics**:
  - *RHS & Matrix Assembly Time*: Time taken to assemble the right-hand side vector and system matrix.
  - *Solve Time*: Time required to solve the linear system using the Conjugate Gradient solver.
- **Tools**: PETSc library configured with parallel execution using MPI.

6.3. Results

Number of Processes	RHS & Matrix Assembly Time (s)	Solve Time (s)	Total Runtime (s
1	0.567762	0.053354	0.621116
2	0.321301	0.075201	0.396502
4	0.159451	0.039462	0.198913
8	0.079609	0.023306	0.102915
16	0.040156	0.013195	0.053351

Table 3: Strong Scaling Performance Results for PETSc CG Solver.

6.4. Log-Log Plot

Below in figure .10is the log-log plot of the total runtime versus the number of processes. (The actual plot should be generated using Python tools such as Matplotlib or another plotting library).

- **X-axis**: Log of the number of processes {1, 2, 4, 8, 16}.
- **Y-axis**: Log of the total runtime.

6.5. Observations

- **Strong Scaling Trend**: The total runtime decreases as the number of processes increases, which indicates that the PETSc solver scales efficiently for the grid size 1024×1024.
- **RHS & Matrix Assembly Time**: The assembly time reduces proportionally with the number of processes, demonstrating good parallelization of the matrix and RHS assembly.
- **Solve Time**: The solve time also decreases with increasing processes, highlighting the efficiency of the PETSc Conjugate Gradient solver under parallel execution.
- **Decreasing Returns**: At higher process counts (e.g., 16 MPI processes), the rate of reduction in runtime diminishes, likely due to communication overhead and limited computational workload per process.

Therefore:

- The PETSc implementation exhibits strong parallel scalability for the given grid size (1024×1024) and performs well with up to 16 MPI processes.
- The total runtime decreases consistently as the number of processes increases, confirming the efficiency of the parallel PETSc Conjugate Gradient solver.
- However, diminishing returns are observed as the number of processes increases, which is expected in parallel computations due to communication overhead.

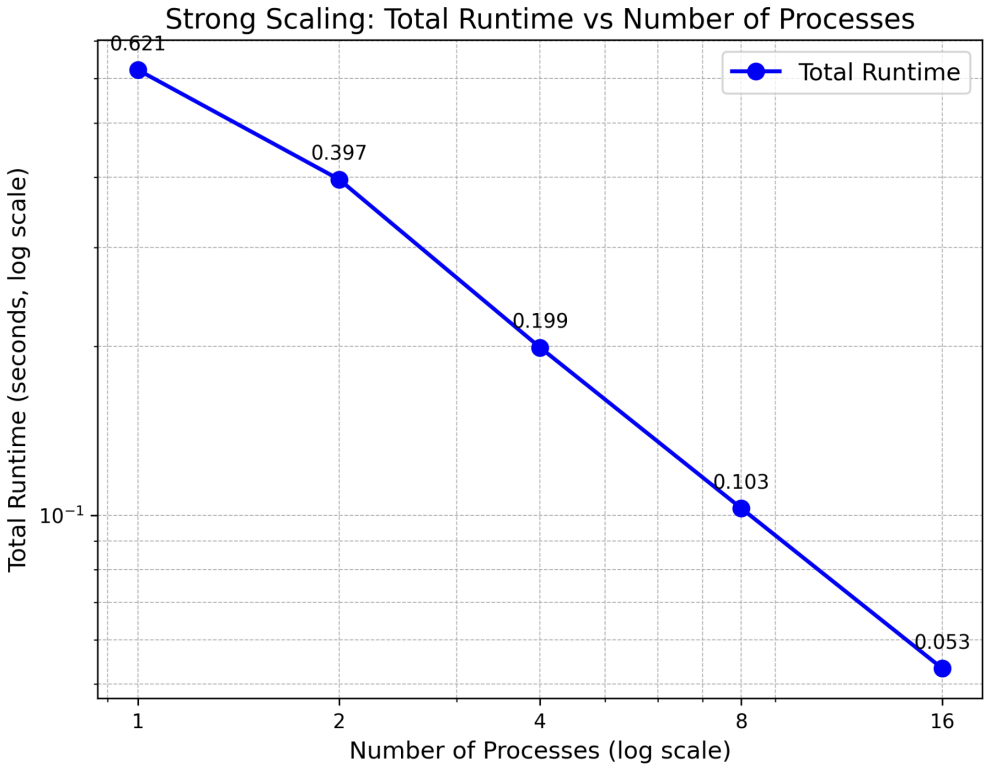

Figure 10: A log log plot for Strong Scaling

References

[1] Ayano Kaneda, Osman Akar, Jingyu Chen, Victoria Alicia Trevino Kala, David Hyde, and Joseph Teran. A deep conjugate direction method for iteratively solving linear systems. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

7. Task: Quality of the Report [15 Points]

Each project will have 100 points (out of which 15 points will be given to the general quality of the written report).

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.), and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi.

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - all the source codes of your solutions;
  - your write-up with your name project_number_lastname_firstname.pdf.
- Submit your .tgz through Icorsi.