Mumps4py: Python interface for the MUMPS solver

Imad Kissami
College of Computing, Mohammed VI Polytechnic University, Morocco

March 31, 2025

1 Introduction

This report describes seven Python scripts demonstrating the MUMPS solver [2, 1] via the `mumps4py` https://github.com/imadki/mumps4py interface. Each example highlights different functionalities, such as centralized and distributed matrix assembly, sparse RHS solving, and elemental matrix formats. The scripts use MPI for parallelism and NumPy for numerical operations. Full Python code with added comments is included for clarity.

2 Example 1: Centralized COO Matrix with Dense RHS

This example solves a linear system $Ax = b$ using a centralized sparse matrix in COO format with a dense RHS.

2.1 Matrix Data

- `irn` (0-based): $[0, 1, 2, 3]$
- `jcn` (0-based): $[0, 1, 2, 3]$
- `a`: $[1.0, 2.0, 3.0, 4.0]$

2.2 Code

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 27 14:08:37 2025

@author: kissami
"""
import numpy as np
from mpi4py import MPI
from scipy.sparse import coo_matrix
from mumps4py.mumps_solver import MumpsSolver
```

```python

# Get MPI rank and size for parallel execution
rank =MPI.COMM_WORLD.Get_rank()
size =MPI.COMM_WORLD.Get_size()

# Set system type to double precision
system ="double"
dtype =np.float32 if system =="single" else np.float64
if system in ["complex64", "complex128"]:
dtype =np.complex64 if system =="complex64" else np.complex128

# Initialize MUMPS solver with double precision and no verbose output
solver =MumpsSolver(verbose=False, system=system)

# Define a 4x4 diagonal matrix in COO format
A = coo_matrix(([1.0, 2.0, 3.0, 4.0], ([0, 1, 2, 3], [0, 1, 2, 3])), shape=(4, 4))

# Set the centralized COO matrix in the solver
solver.set_coo_centralized(A)

# Define the right-hand side vector
rhs =np.array([1.0, 2.0, 3.0, 4.0])
solver.set_rhs_centralized(rhs)

# Perform analysis phase (symbolic factorization)
solver.analyze()
# Perform numerical factorization
solver.factorize()
# Solve the system, overwriting rhs with the solution
solver.solve()

# Print the solution on all ranks
print("Solution is:", rhs)
```

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Expected solution: $x = [1, 1, 1, 1]$.

3 Example 2: Distributed COO Matrix with Dense RHS

This example uses a distributed COO matrix with a dense RHS.

3.1 Matrix Data

- `irn` (0-based): $[0, 1, 3, 4, 1, 0, 4, 2, 1, 2, 0, 2]$
- `jcn` (0-based): $[1, 2, 2, 4, 0, 0, 1, 3, 4, 1, 2, 2]$
- `a`: $[3.0, -3.0, 2.0, 1.0, 3.0, 2.0, 4.0, 2.0, 6.0, -1.0, 4.0, 1.0]$

3.2 Code

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import numpy as np
from mpi4py import MPI
from mumps4py.mumps_solver import MumpsSolver

# MPI setup
rank =MPI.COMM_WORLD.Get_rank()
size =MPI.COMM_WORLD.Get_size()

# Set double precision
system ="double"
dtype =np.float32 if system =="single" else np.float64
if system in ["complex64", "complex128"]:
dtype =np.complex64 if system =="complex64" else np.complex128

# Initialize solver
solver =MumpsSolver(verbose=False, system=system)

# Define matrix size on rank 0 only
if rank ==0:
    n = 5
else:
    n = None

# Define COO matrix data (0-based indices)
irn =np.array([0,1,3,4,1,0,4,2,1,2,0,2], dtype=np.int32)
jcn =np.array([1,2,2,4,0,0,1,3,4,1,2,2], dtype=np.int32)
a = np.array([3.0,-3.0,2.0,1.0,3.0,2.0,4.0,2.0,6.0,-1.0,4.0,1.0], dtype=dtype)
b = np.array([[20.0,24.0,9.0,6.0,13.0],[20.0,24.0,9.0,6.0,13.0]], dtype=dtype)

# Split matrix entries across MPI processes
indices =np.arange(len(irn))
split_indices =np.array_split(indices, size)
local_indices =split_indices[rank]

# Extract local portions of the matrix
local_irn =irn[local_indices]
local_jcn =jcn[local_indices]
local_a =a[local_indices]

# Set distributed matrix (convert to 1-based indices for MUMPS)
solver.set_rcd_distributed(local_irn+1, local_jcn+1, local_a, n)
solver.set_icntl(18,3) # Enable distributed assembly

# Analyze and factorize the matrix
solver.analyze()
solver.factorize()

# Set RHS on rank 0 only
if MPI.COMM_WORLD.Get_rank() ==0:
solver.set_rhs_centralized(b)

# Solve the system
solver.solve()
if rank ==0:
print("Solution 3", b)
```

$$A = \begin{bmatrix} 2 & 3 & 4 & 0 & 0 \\ 3 & -1 & -3 & 0 & 6 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

4 Example 3: Distributed COO Matrix with Distributed Solution

This example extends Example 2 with a distributed solution.

4.1 Matrix Data

- `irn`: $[0, 1, 3, 4, 1, 0, 4, 2, 1, 2, 0, 2]$
- `jcn`: $[1, 2, 2, 4, 0, 0, 1, 3, 4, 1, 2, 2]$
- `a`: $[3.0, -3.0, 2.0, 1.0, 3.0, 2.0, 4.0, 2.0, 6.0, -1.0, 4.0, 1.0]$

4.2 Code

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 27 14:23:15 2025

@author: kissami
"""
from mumps4py.mumps_solver import MumpsSolver
import numpy as np
from mpi4py import MPI

# MPI initialization
comm =MPI.COMM_WORLD
rank =comm.Get_rank()
size =comm.Get_size()

# Set double precision
system ="double"
dtype =np.float32 if system =="single" else np.float64
if system in ["complex64", "complex128"]:
dtype =np.complex64 if system =="complex64" else np.complex128

# Initialize solver
solver =MumpsSolver(verbose=False, system=system)

# Define matrix size
n = 5
# Define COO matrix (1-based indices for MUMPS)
irn =np.array([0,1,3,4,1,0,4,2,1,2,0,2], dtype=np.int32) +1
jcn =np.array([1,2,2,4,0,0,1,3,4,1,2,2], dtype=np.int32) +1
a = np.array([3.0,-3.0,2.0,1.0,3.0,2.0,4.0,2.0,6.0,-1.0,4.0,1.0], dtype=dtype)
b = np.array([20.0,24.0,9.0,6.0,13.0], dtype=dtype)

# Distribute matrix entries across processes
```

```
indices =np.arange(len(irn))
split_indices =np.array_split(indices, size)
local_indices =split_indices[rank]

# Local matrix portions
local_irn =irn[local_indices]
local_jcn =jcn[local_indices]
local_a =a[local_indices]

# Configure solver for distributed assembly and solution
solver.set_icntl(18, 3) # Distributed matrix input
solver.set_rcd_distributed(local_irn, local_jcn, local_a, n)
solver.set_icntl(21, 1) # Distributed solution output

# Analyze and factorize
solver.analyze()
solver.factorize()

# Set RHS on rank 0
if rank ==0:
solver.set_rhs_centralized(b)

# Enable and compute distributed solution
solver.enable_distributed_solution(1)
solver.solve()

# Retrieve distributed solution
shape =b.shape
dtype =np.float64
distributed_solution =solver.pointer_to_numpy(solver.struct.sol_loc, dtype, shape)
print("Distributed solution :", distributed_solution)

# Get solution indices
isol_indices =solver.pointer_to_numpy(solver.struct.isol_loc, np.int32, b.shape)
print("Solution indices :", isol_indices)

# Reconstruct full solution
final_solution =np.zeros(n, dtype=dtype)
final_solution[isol_indices -1] =distributed_solution # Adjust for 0-based indexing
print("Final solution :", final_solution)
```

5 Example 4: Elemental Matrix Format

This example uses an elemental matrix format for a complex-valued system.

5.1 Matrix Data

- `eltptr` (1-based): $[1, 4, 7]$
- `eltvar` (1-based): $[1, 2, 3, 3, 4, 5]$
- `a_elt`: $[-1, 2, 1, 2, 1, 1, 3, 1, 1, 2, 1, 3, -1, 2, 2, 3, -1, 1]$

5.2 Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import numpy as np
```

```python
from mpi4py import MPI
from mumps4py.mumps_solver import MumpsSolver

# MPI setup
rank =MPI.COMM_WORLD.Get_rank()
size =MPI.COMM_WORLD.Get_size()

# Set complex128 precision
system ="complex128"
dtype =np.float32 if system =="single" else np.float64
if system in ["complex64", "complex128"]:
dtype =np.complex64 if system =="complex64" else np.complex128

# Initialize solver
solver =MumpsSolver(verbose=False, system=system)

# Define matrix parameters
n = 5  # Matrix order
nelt =2  # Number of elements

# Elemental matrix data (1-based)
eltptr =np.array([1, 4, 7], dtype=np.int32) # Element pointers
eltvar =np.array([1, 2, 3, 3, 4, 5], dtype=np.int32) # Variable indices
a_elt =np.array([-1, 2, 1, 2, 1, 1, 3, 1, 1, 2, 1, 3, -1, 2, 2, 3, -1, 1], dtype=dtype) # Values
bb = np.array([1, 20, 3, 4, 5], dtype=dtype) # RHS
rhs = bb.copy() # Copy for verification

# Configure solver for elemental format
solver.set_icntl(5, 1) # Use elemental matrix format
solver.set_icntl(18, 0) # Centralized assembly

# Set matrix and RHS
solver.set_elemental_matrix(eltptr, eltvar, a_elt, n, nelt)
solver.set_rhs_centralized(bb)

# Solve the system
solver.analyze()
solver.factorize()
solver.solve()

# Print solution on rank 0
if rank ==0:
print("Solution:", bb)

# Define A for verification (not in original code)
A = np.array([[-1, 2, 3, 0, 0], [2, 1, 1, 0, 0], [1, 1, 3, -1, 3], [0, 0, 1, 2, -1], [0, 0, 3, 2, 1]], dtype=dtype)
print("Check solution:", A.dot(bb) -rhs)
```

$$A = \begin{bmatrix} -1 & 2 & 1 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 1 & 1 & 3 & -1 & 3 \\ 0 & 0 & 1 & 2 & -1 \\ 0 & 0 & 3 & 2 & 1 \end{bmatrix}$$

6 Example 5: Centralized COO Matrix with Manual Job Calls and Reused Factorization

This example uses manual MUMPS job calls to solve a 5×5 system and illustrates that if the matrix structure (`irn`, `jcn`) and values (`a`) remain unchanged, the factorization step can

be reused for multiple RHS vectors, avoiding redundant computation.

6.1 Matrix Data

- `irn`: $[0, 1, 3, 4, 1, 0, 4, 2, 1, 2, 0, 2]$
- `jcn`: $[1, 2, 2, 4, 0, 0, 1, 3, 4, 1, 2, 2]$
- `a`: $[3.0, -3.0, 2.0, 1.0, 3.0, 2.0, 4.0, 2.0, 6.0, -1.0, 4.0, 1.0]$

6.2 Code

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import numpy as np
from mpi4py import MPI
from mumps4py.mumps_solver import MumpsSolver

# MPI setup
rank =MPI.COMM_WORLD.Get_rank()
size =MPI.COMM_WORLD.Get_size()

# Set double precision
system ="double"
dtype =np.float32 if system =="single" else np.float64
if system in ["complex64", "complex128"]:
dtype =np.complex64 if system =="complex64" else np.complex128

# Initialize solver
solver =MumpsSolver(verbose=False, system=system)

# Define matrix and two different RHS vectors
n = 5
irn =np.array([0,1,3,4,1,0,4,2,1,2,0,2], dtype=np.int32)
jcn =np.array([1,2,2,4,0,0,1,3,4,1,2,2], dtype=np.int32)
a = np.array([3.0,-3.0,2.0,1.0,3.0,2.0,4.0,2.0,6.0,-1.0,4.0,1.0], dtype=dtype)
b1 = np.array([20.0, 24.0, 9.0, 6.0, 13.0], dtype=dtype) # First RHS
b2 = np.array([1.0, 2.0, 3.0, 4.0, 5.0], dtype=dtype) # Second RHS

n = len(b1) # Matrix size from RHS

# Start timing
ts = MPI.Wtime()

# Set centralized matrix (1-based indices)
solver.set_rcd_centralized(irn+1, jcn+1, a, n)
solver._mumps_call(job=1) # Analysis phase (symbolic factorization)

# Factorize once (numerical factorization)
solver._mumps_call(job=2) # Factorization phase, done only once

# Solve for first RHS
rhs1 =b1.copy() # Copy to preserve original
solver.set_rhs_centralized(rhs1)
solver._mumps_call(3) # Solve phase
if rank ==0:
print("Solution for b1:", rhs1)

# Solve for second RHS reusing factorization
rhs2 =b2.copy() # Copy to preserve original
solver.set_rhs_centralized(rhs2)
solver._mumps_call(3) # Solve phase, no need to re-factorize
```

```
if rank ==0:
print("Solution for b2:", rhs2)

# Print total CPU time
if rank ==0:
print("CPU time is ", MPI.Wtime() -ts)
```

6.3 Explanation

The matrix A is defined once, and its factorization (`job=2`) is performed only once after analysis (`job=1`). Two different RHS vectors, $b_1 = [20, 24, 9, 6, 13]$ and $b_2 = [1, 2, 3, 4, 5]$, are solved using the same factorization by calling only the solve phase (`job=3`) for each. This demonstrates that as long as `irn`, `jcn`, and `a` do not change, re-factorization is unnecessary, optimizing performance.

7 Example 6: Updating Centralized Matrix Values

This example updates matrix values without re-analysis.

7.1 Matrix Data

- `irn`: $[0, 1, 3, 4, 1, 0, 4, 2, 1, 2, 0, 2]$
- `jcn`: $[1, 2, 2, 4, 0, 0, 1, 3, 4, 1, 2, 2]$
- Initial `a`: $[3.0, -3.0, 2.0, 1.0, 3.0, 2.0, 4.0, 2.0, 6.0, -1.0, 4.0, 1.0]$
- Updated `a`: $[6.0, -33.0, 2.0, 1.0, 33.0, 22.0, 41.0, 2.0, 66.0, -11.0, 4.0, 1.0]$

7.2 Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 5 11:23:56 2025

@author: kissami
"""
from scipy.sparse import coo_matrix
from mumps4py.mumps_solver import MumpsSolver
import numpy as np
from mpi4py import MPI

# MPI setup
rank =MPI.COMM_WORLD.Get_rank()
size =MPI.COMM_WORLD.Get_size()

# Set double precision
system ="double"
dtype =np.float32 if system =="single" else np.float64
if system in ["complex64", "complex128"]:
dtype =np.complex64 if system =="complex64" else np.complex128

```

```python
# Initialize solver
solver =MumpsSolver(verbose=False, system=system)

# Define initial matrix and RHS
n = 5
irn =np.array([0,1,3,4,1,0,4,2,1,2,0,2], dtype=np.int32)
jcn =np.array([1,2,2,4,0,0,1,3,4,1,2,2], dtype=np.int32)
a = np.array([3.0,-3.0,2.0,1.0,3.0,2.0,4.0,2.0,6.0,-1.0,4.0,1.0], dtype=dtype)
b = np.array([20.0,24.0,9.0,6.0,13.0], dtype=dtype)

# Set initial matrix (1-based)
solver.set_rcd_centralized(irn+1, jcn+1, a, n)
solver._mumps_call(job=1) # Analyze structure

# Solve with initial values
rhs =b.copy()
solver._mumps_call(job=2) # Factorize
solver.set_rhs_centralized(rhs)
solver._mumps_call(3) # Solve
print("Solution:", rhs)

# Update matrix values
a = np.array([6.0,-33.0,2.0,1.0,33.0,22.0,41.0,2.0,66.0,-11.0,4.0,1.0], dtype=dtype)
solver.set_data_centralized(a, n) # Update values only

# Solve with updated values
rhs =b.copy()
solver._mumps_call(job=2) # Re-factorize
solver.set_rhs_centralized(rhs)
solver._mumps_call(3) # Re-solve
print("Solution:", rhs)

# Verify solution
n = 5
A = coo_matrix((a, (irn, jcn)), shape=(n, n))
print("check the solution:", A.dot(rhs), b)
```

8 Example 7: Sparse RHS with Centralized COO Matrix

This example solves a system with a sparse RHS.

8.1 Matrix Data

- `irn`: $[0, 0, 1, 1, 1, 2, 2, 3, 3]$
- `jcn`: $[0, 1, 0, 1, 2, 1, 2, 2, 3]$
- `a`: $[4.0, -1.0, -1.0, 4.0, -1.0, -1.0, 4.0, -1.0, 3.0]$

8.2 Sparse RHS Data

- `rhs_values`: $[1.1, 2.2, 3.1, 4.1, 3.2]$
- `rhs_row_indices` (1-based): $[1, 3, 4, 2, 3]$
- `rhs_col_ptr` (1-based): $[1, 4, 6]$

8.3 Code

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 27 14:08:37 2025

@author: kissami
"""
import numpy as np
from mpi4py import MPI
from mumps4py.mumps_solver import MumpsSolver

# MPI setup
rank =MPI.COMM_WORLD.Get_rank()
size =MPI.COMM_WORLD.Get_size()

# Set double precision
system ="double"
dtype =np.float32 if system =="single" else np.float64
if system in ["complex64", "complex128"]:
dtype =np.complex64 if system =="complex64" else np.complex128

# Initialize solver
solver =MumpsSolver(verbose=False, system=system)

# Define 4x4 matrix
n = 4
nnz =9
irn =np.array([0, 0, 1, 1, 1, 2, 2, 3, 3], dtype=np.int32)
jcn =np.array([0, 1, 0, 1, 2, 1, 2, 2, 3], dtype=np.int32)
a = np.array([4.0, -1.0, -1.0, 4.0, -1.0, -1.0, 4.0, -1.0, 3.0], dtype=dtype)

# Set centralized matrix (1-based)
solver.set_rcd_centralized(irn+1, jcn+1, a, n)
solver.analyze()
solver.factorize()

# Configure for sparse RHS
solver.set_icntl(20, 1) # Enable sparse RHS

# Define sparse RHS parameters
nz_rhs =5  # Number of non-zeros
nrhs =2    # Number of RHS columns
rhs_values =np.array([1.1, 2.2, 3.1, 4.1, 3.2], dtype=dtype) # Non-zero values
rhs_row_indices =np.array([1, 3, 4, 2, 3], dtype=np.int32) # Row indices (1-based)
rhs_col_ptr =np.array([1, 4, 6], dtype=np.int32) # Column pointers (1-based)

# Initialize dense RHS storage on rank 0
if rank ==0:
rhs =np.zeros((2,4))
solver.set_rhs_centralized(rhs)

# Set sparse RHS
solver.set_rhs_sparse(rhs_values, rhs_row_indices, rhs_col_ptr, nrhs)
solver.solve()

# Print solution
print("Solution is :", rhs)
```

$$A = \begin{bmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ 0 & 0 & -1 & 3 \end{bmatrix}$$

9 Benchmarks: Python vs C

In this section, we compare the runtime of Python and C (through Cython wrapper) using MUMPS-5.3.5 (pt-scotch, without OpenMP) for various matrix sizes obtained from Finite Volume discretization using Manapy [3]. We attempt to determine if the Cython-based interface imposes any extra computational cost over the pure C implementation.

Figures 1 and 3 show the performance of the 2D and 3D problems, respectively. The x-axis is the configuration (matrix size and number of processes), and the y-axis (log scale) is the execution time in seconds for analysis, factorization, and solution phases.

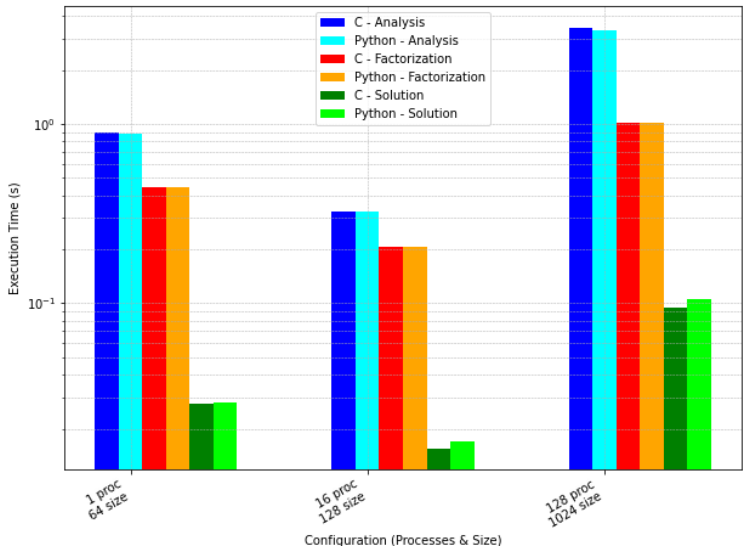

Figure 1: Timing (s) using MUMPS for all steps through C vs Python for 2D matrices

Key Observations:

- All configurations tested have Python (through Cython) performing as well as C.
- There is no extra CPU cost of calling the Cython wrapper, i.e., the Python function calls are just as efficient as direct C function calls. This validates that Cython removes Python overhead by compiling Python code to C and providing almost native call speed in calling MUMPS.

Figures 2 and 4 illustrate the speedup achieved when using the Python-based MUMPS wrapper compared to the native C implementation for 2D and 3D matrices, respectively.

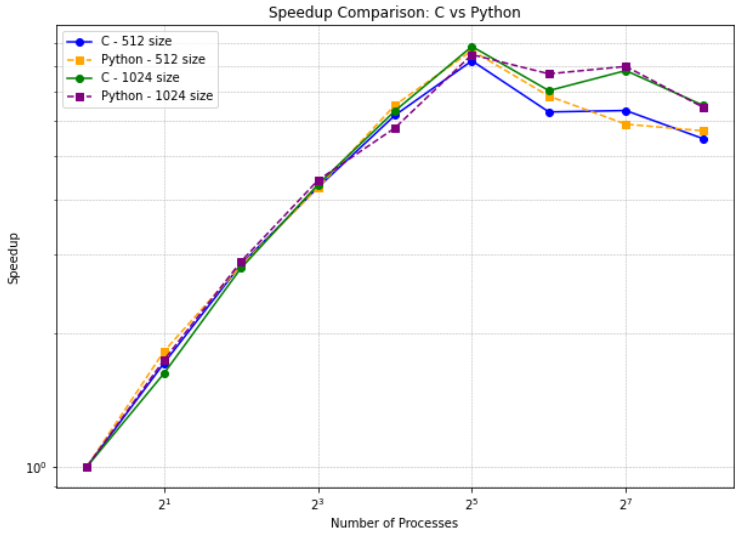

Figure 2: Speedup using MUMPS for all steps through C vs Python for 2D matrices

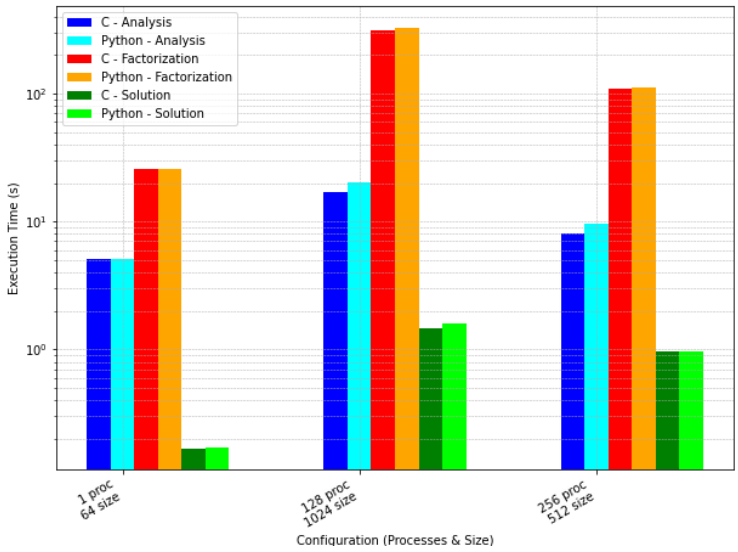

Figure 3: Timing (s) using MUMPS for all steps through C vs Python for 3D matrices

10 Conclusion

This report has illustrated the application of MUMPS through the mumps4py interface to solve sparse linear systems in parallel computing. Various examples have been presented, spanning various matrix assembly methods (centralized and distributed), right-hand side (RHS) management (dense and sparse), and factor reuse for enhancing performance.

The benchmarking outcomes indicate that the Python MUMPS interface (through Cython)

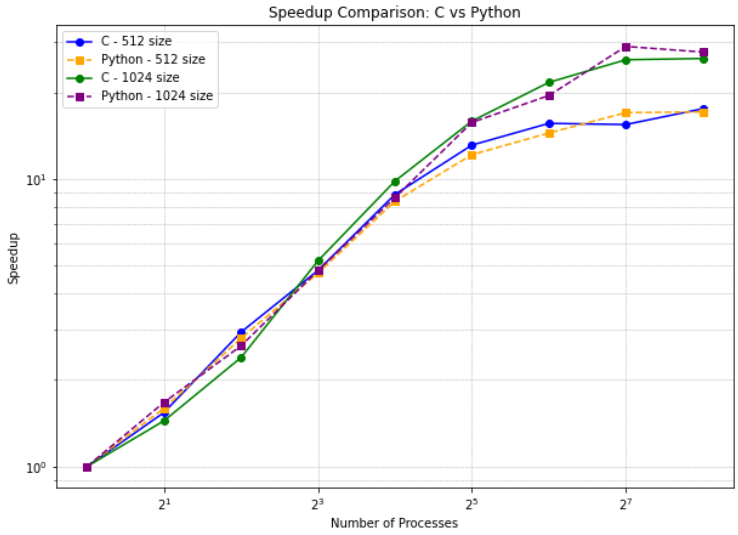

Figure 4: Speedup using MUMPS for all steps through C vs Python for 3D matrices

is equally fast as the native C code with little or no overhead. This confirms the effectiveness of the Python wrapper and its suitability for high-performance computing (HPC) applications. The tests also demonstrate the scalability of MUMPS along with its efficiency in handling big sparse matrices.

In summary, mumps4py offers an easy-to-use and flexible interface to call MUMPS from Python and is a great software package for scientists and engineers dealing with large-scale scientific computations. GPU acceleration and more optimizations on heterogeneous computing architectures can be investigated in future work.

References

[1] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Transactions on Mathematical Software*, 45(1):2:1–2:26, 2019.

[2] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[3] Imad Kissami. Manapy: an mpi-based python framework for solving poisson's equation using finite volume on unstructured-grid. In *AIP Conference Proceedings*, volume 3034. AIP Publishing, 2024.