

MUMPS Solver Examples with mumps4py

March 09, 2025

1 Introduction

This report describes seven Python scripts demonstrating the MUMPS solver via the `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

```
#!/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

# Get MPI rank and size for parallel execution
```

```

34 rank =MPI.COMM_WORLD.Get_rank()
35 size =MPI.COMM_WORLD.Get_size()
36
37 # Set system type to double precision
38 system ="double"
39 dtype =np.float32 if system =="single" else np.float64
40 if system in ["complex64", "complex128"]:
41     dtype =np.complex64 if system =="complex64" else np.complex128
42
43 # Initialize MUMPS solver with double precision and no verbose output
44 solver =MumpsSolver(verbose=False, system=system)
45
46 # Define a 4x4 diagonal matrix in COO format
47 A =coo_matrix(([1.0, 2.0, 3.0, 4.0], ([0, 1, 2, 3], [0, 1, 2, 3])), shape=(4, 4))
48
49 # Set the centralized COO matrix in the solver
50 solver.set_coo_centralized(A)
51
52 # Define the right-hand side vector
53 rhs =np.array([1.0, 2.0, 3.0, 4.0])
54 solver.set_rhs_centralized(rhs)
55
56 # Perform analysis phase (symbolic factorization)
57 solver.analyze()
58 # Perform numerical factorization
59 solver.factorize()
60 # Solve the system, overwriting rhs with the solution
61 solver.solve()
62
63 # Print the solution on all ranks
64 print("Solution is:", rhs)

```

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

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

67 3 Example 2: Distributed COO Matrix with Dense 68 RHS

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

70 3.1 Matrix Data

- 71 • `irn` (0-based): `[0, 1, 3, 4, 1, 0, 4, 2, 1, 2, 0, 2]`
- 72 • `jcn` (0-based): `[1, 2, 2, 4, 0, 0, 1, 3, 4, 1, 2, 2]`
- 73 • `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]`

74 3.2 Code

```

76
77 #!/usr/bin/env python3
78 # -*- coding: utf-8 -*-
79 import numpy as np
80 from mpi4py import MPI
81 from mumps4py.mumps_solver import MumpsSolver
82
83 # MPI setup
84 rank =MPI.COMM_WORLD.Get_rank()
85 size =MPI.COMM_WORLD.Get_size()
86
87 # Set double precision
88 system = "double"
89 dtype =np.float32 if system == "single" else np.float64
90 if system in ["complex64", "complex128"]:
91     dtype =np.complex64 if system == "complex64" else np.complex128
92
93 # Initialize solver
94 solver =MumpsSolver(verbose=False, system=system)
95
96 # Define matrix size on rank 0 only
97 if rank ==0:
98     n = 5
99 else:
100     n = None
101
102 # Define COO matrix data (0-based indices)
103 irn =np.array([0,1,3,4,1,0,4,2,1,2,0,2], dtype=np.int32)
104 jcn =np.array([1,2,2,4,0,0,1,3,4,1,2,2], dtype=np.int32)
105 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)
106 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)
107
108 # Split matrix entries across MPI processes
109 indices =np.arange(len(irn))
110 split_indices =np.array_split(indices, size)
111 local_indices =split_indices[rank]
112
113 # Extract local portions of the matrix
114 local_irn =irn[local_indices]
115 local_jcn =jcn[local_indices]
116 local_a =a[local_indices]
117
118 # Set distributed matrix (convert to 1-based indices for MUMPS)
119 solver.set_rcd_distributed(local_irn+1, local_jcn+1, local_a, n)
120 solver.set_icntl(18,3) # Enable distributed assembly
121
122 # Analyze and factorize the matrix
123 solver.analyze()
124 solver.factorize()
125
126 # Set RHS on rank 0 only
127 if MPI.COMM_WORLD.Get_rank() ==0:
128     solver.set_rhs_centralized(b)
129
130 # Solve the system
131 solver.solve()
132 if rank ==0:
133     print("Solution 3", b)
134

```

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

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

```
#!/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
```

```

190 solver.set_rcd_distributed(local_irn, local_jcn, local_a, n)
191 solver.set_icntl(21, 1) # Distributed solution output
192
193 # Analyze and factorize
194 solver.analyze()
195 solver.factorize()
196
197 # Set RHS on rank 0
198 if rank == 0:
199     solver.set_rhs_centralized(b)
200
201 # Enable and compute distributed solution
202 solver.enable_distributed_solution(1)
203 solver.solve()
204
205 # Retrieve distributed solution
206 shape = b.shape
207 dtype = np.float64
208 distributed_solution = solver.pointer_to_numpy(solver.struct.sol_loc, dtype, shape)
209 print("Distributed solution :", distributed_solution)
210
211 # Get solution indices
212 isol_indices = solver.pointer_to_numpy(solver.struct.isol_loc, np.int32, b.shape)
213 print("Solution indices :", isol_indices)
214
215 # Reconstruct full solution
216 final_solution = np.zeros(n, dtype=dtype)
217 final_solution[isol_indices - 1] = distributed_solution # Adjust for 0-based indexing
218 print("Final solution :", final_solution)
219

```

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

```

228 #!/usr/bin/env python3
229 # -*- coding: utf-8 -*-
230 import numpy as np
231 from mpi4py import MPI
232 from mumps4py.mumps_solver import MumpsSolver
233
234 # MPI setup
235 rank = MPI.COMM_WORLD.Get_rank()
236 size = MPI.COMM_WORLD.Get_size()
237
238 # Set complex128 precision
239 system = "complex128"
240 dtype = np.float32 if system == "single" else np.float64
241 if system in ["complex64", "complex128"]:

```

```

243 dtype =np.complex64 if system =="complex64" else np.complex128
244
245 # Initialize solver
246 solver =MumpsSolver(verbose=False, system=system)
247
248 # Define matrix parameters
249 n =5 # Matrix order
250 nelt =2 # Number of elements
251
252 # Elemental matrix data (1-based)
253 eltptr =np.array([1, 4, 7], dtype=np.int32) # Element pointers
254 eltvar =np.array([1, 2, 3, 3, 4, 5], dtype=np.int32) # Variable indices
255 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
256 bb =np.array([1, 20, 3, 4, 5], dtype=dtype) # RHS
257 rhs =bb.copy() # Copy for verification
258
259 # Configure solver for elemental format
260 solver.set_icntl(5, 1) # Use elemental matrix format
261 solver.set_icntl(18, 0) # Centralized assembly
262
263 # Set matrix and RHS
264 solver.set_elemental_matrix(eltptr, eltvar, a_elt, n, nelt)
265 solver.set_rhs_centralized(bb)
266
267 # Solve the system
268 solver.analyze()
269 solver.factorize()
270 solver.solve()
271
272 # Print solution on rank 0
273 if rank ==0:
274     print("Solution:", bb)
275
276 # Define A for verification (not in original code)
277 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)
278 print("Check solution:", A.dot(bb) -rhs)
279

```

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

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]

288 • 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]

289 6.2 Code

```
290
291 #!/usr/bin/env python3
292 # -*- coding: utf-8 -*-
293 import numpy as np
294 from mpi4py import MPI
295 from mumps4py.mumps_solver import MumpsSolver
296
297 # MPI setup
298 rank =MPI.COMM_WORLD.Get_rank()
299 size =MPI.COMM_WORLD.Get_size()
300
301 # Set double precision
302 system ="double"
303 dtype =np.float32 if system =="single" else np.float64
304 if system in ["complex64", "complex128"]:
305     dtype =np.complex64 if system =="complex64" else np.complex128
306
307 # Initialize solver
308 solver =MumpsSolver(verbose=False, system=system)
309
310 # Define matrix and two different RHS vectors
311 n =5
312 irn =np.array([0,1,3,4,1,0,4,2,1,2,0,2], dtype=np.int32)
313 jcn =np.array([1,2,2,4,0,0,1,3,4,1,2,2], dtype=np.int32)
314 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)
315 b1 =np.array([20.0, 24.0, 9.0, 6.0, 13.0], dtype=dtype) # First RHS
316 b2 =np.array([1.0, 2.0, 3.0, 4.0, 5.0], dtype=dtype) # Second RHS
317
318 n =len(b1) # Matrix size from RHS
319
320 # Start timing
321 ts =MPI.Wtime()
322
323 # Set centralized matrix (1-based indices)
324 solver.set_rcd_centralized(irn+1, jcn+1, a, n)
325 solver._mumps_call(job=1) # Analysis phase (symbolic factorization)
326
327 # Factorize once (numerical factorization)
328 solver._mumps_call(job=2) # Factorization phase, done only once
329
330 # Solve for first RHS
331 rhs1 =b1.copy() # Copy to preserve original
332 solver.set_rhs_centralized(rhs1)
333 solver._mumps_call(3) # Solve phase
334 if rank ==0:
335     print("Solution for b1:", rhs1)
336
337 # Solve for second RHS reusing factorization
338 rhs2 =b2.copy() # Copy to preserve original
339 solver.set_rhs_centralized(rhs2)
340 solver._mumps_call(3) # Solve phase, no need to re-factorize
341 if rank ==0:
342     print("Solution for b2:", rhs2)
343
344 # Print total CPU time
345 if rank ==0:
346     print("CPU time is ", MPI.Wtime() -ts)
347
```

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

# 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)
```



```

397 # Set initial matrix (1-based)
398 solver.set_rcd_centralized(irn+1, jcn+1, a, n)
399 solver._mumps_call(job=1) # Analyze structure
400
401 # Solve with initial values
402 rhs = b.copy()
403 solver._mumps_call(job=2) # Factorize
404 solver.set_rhs_centralized(rhs)
405 solver._mumps_call(3) # Solve
406 print("Solution:", rhs)
407
408 # Update matrix values
409 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)
410 solver.set_data_centralized(a, n) # Update values only
411
412 # Solve with updated values
413 rhs = b.copy()
414 solver._mumps_call(job=2) # Re-factorize
415 solver.set_rhs_centralized(rhs)
416 solver._mumps_call(3) # Re-solve
417 print("Solution:", rhs)
418
419 # Verify solution
420 n = 5
421 A = coo_matrix((a, (irn, jcn)), shape=(n, n))
422 print("check the solution:", A.dot(rhs), b)
423

```

424 8 Example 7: Sparse RHS with Centralized COO Ma- 425 trix

426 This example solves a system with a sparse RHS.

427 8.1 Matrix Data

- 428 • irn: [0, 0, 1, 1, 1, 2, 2, 3, 3]
- 429 • jcn: [0, 1, 0, 1, 2, 1, 2, 2, 3]
- 430 • a: [4.0, -1.0, -1.0, 4.0, -1.0, -1.0, 4.0, -1.0, 3.0]

431 8.2 Sparse RHS Data

- 432 • rhs_values: [1.1, 2.2, 3.1, 4.1, 3.2]
- 433 • rhs_row_indices (1-based): [1, 3, 4, 2, 3]
- 434 • rhs_col_ptr (1-based): [1, 4, 6]

435 8.3 Code

```

436 #!/usr/bin/env python3
437 # -*- coding: utf-8 -*-
438 """
439
440

```

```

441 Created on Thu Feb 27 14:08:37 2025
442
443 @author: kissami
444 """
445 import numpy as np
446 from mpi4py import MPI
447 from mumps4py.mumps_solver import MumpsSolver
448
449 # MPI setup
450 rank =MPI.COMM_WORLD.Get_rank()
451 size =MPI.COMM_WORLD.Get_size()
452
453 # Set double precision
454 system ="double"
455 dtype =np.float32 if system =="single" else np.float64
456 if system in ["complex64", "complex128"]:
457     dtype =np.complex64 if system =="complex64" else np.complex128
458
459 # Initialize solver
460 solver =MumpsSolver(verbose=False, system=system)
461
462 # Define 4x4 matrix
463 n =4
464 nnz =9
465 irn =np.array([0, 0, 1, 1, 1, 2, 2, 3, 3], dtype=np.int32)
466 jcn =np.array([0, 1, 0, 1, 2, 1, 2, 2, 3], dtype=np.int32)
467 a =np.array([4.0, -1.0, -1.0, 4.0, -1.0, -1.0, 4.0, -1.0, 3.0], dtype=dtype)
468
469 # Set centralized matrix (1-based)
470 solver.set_rcd_centralized(irn+1, jcn+1, a, n)
471 solver.analyze()
472 solver.factorize()
473
474 # Configure for sparse RHS
475 solver.set_icntl(20, 1) # Enable sparse RHS
476
477 # Define sparse RHS parameters
478 nz_rhs =5 # Number of non-zeros
479 nrhs =2 # Number of RHS columns
480 rhs_values =np.array([1.1, 2.2, 3.1, 4.1, 3.2], dtype=dtype) # Non-zero values
481 rhs_row_indices =np.array([1, 3, 4, 2, 3], dtype=np.int32) # Row indices (1-based)
482 rhs_col_ptr =np.array([1, 4, 6], dtype=np.int32) # Column pointers (1-based)
483
484 # Initialize dense RHS storage on rank 0
485 if rank ==0:
486     rhs =np.zeros((2,4))
487     solver.set_rhs_centralized(rhs)
488
489 # Set sparse RHS
490 solver.set_rhs_sparse(rhs_values, rhs_row_indices, rhs_col_ptr, nrhs)
491 solver.solve()
492
493 # Print solution
494 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 [?]. 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.

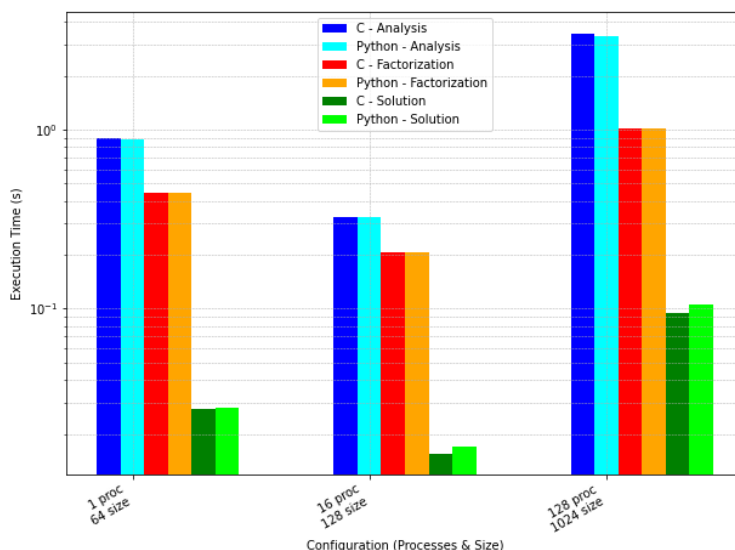


Figure 1: Timing (s) using MUMPS through C vs through Python for 2D matrices

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.

10 Conclusion

This report has demonstrated the use of MUMPS via the mumps4py interface for solving sparse linear systems in a parallel computing environment. Several examples were presented,

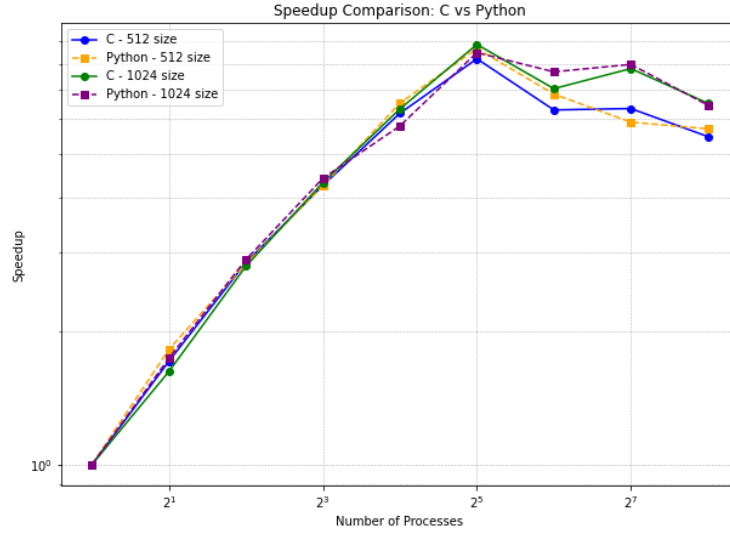


Figure 2: Speedup using MUMPS through C vs through Python for 2D matrices

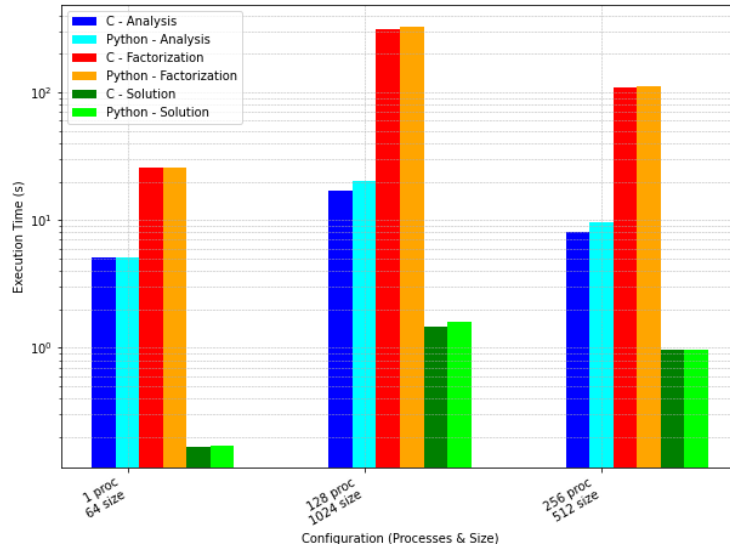


Figure 3: Timing (s) using MUMPS through C vs through Python for 3D matrices

covering different matrix assembly techniques (centralized and distributed), right-hand side (RHS) handling (dense and sparse), and factorization reuse to optimize performance.

The benchmarking results show that the Python-based MUMPS interface (via Cython) achieves performance comparable to the native C implementation, with negligible overhead. This validates the efficiency of the Python wrapper and confirms its suitability for high-performance computing (HPC) applications. The experiments also highlight the scalability of MUMPS and its ability to handle large sparse matrices efficiently.

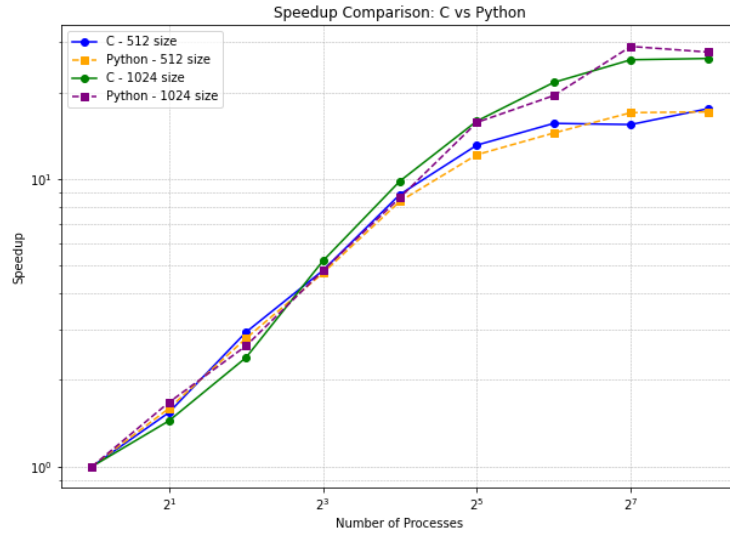


Figure 4: Speedup using MUMPS through C vs through Python for 3D matrices

Overall, mumps4py provides a convenient and flexible interface for leveraging MUMPS within Python, making it an excellent choice for researchers and engineers working on large-scale scientific computations. Future work could explore GPU acceleration and further optimizations for heterogeneous computing environments.