analysis

April 4, 2025

1 Project Milestone 05 (NE 591) - Parallel Diffusion Code

- **Author**: Kirill Shumilov
- **Date**: Fri, April 4, 2025

[183]:
```
import pandas as pd
import matplotlib.pyplot as plt
import pathlib

analysis_dir = pathlib.Path('.').absolute()
examples_dir = analysis_dir.parent / 'examples'
tests_dir = analysis_dir / 'tests'
executable = analysis_dir.parent / 'shumilov_project02'
```

2 1. Description of Work

In this project milestone, we implement Parallel Point-Jacobi, Gauss-Seidel, and Successive-Over-Relaxation algorithm to solving a finite-difference 2D Neutron Diffusion problem in a non-multiplying medium. To acheive this task, two major ideas are used:

1. Partitioning of the source/flux matrices among processes

Both the source and the flux matrices are partioned into blocks of appropriate size and distributed among processes, arranged in a 2D cartesian topology. At each iteration, each process updates its flux block according to serial PJ/GS/SOR algorithm (baring complication with GS/SOR). At the end of each iteration, each process distributes its halo, a region of single-cell wide among the neigbhoring cells. The distribution is done in non-blocking point-to-point Send/Recv. Each process, queues their sends and receives, and then waits for the data transfers to complete. This allows to minimize synchronization between the processes.

For GS/SOR algorithm, simple serial version of the iteration is not possible. Therefore, to restore parallelism, each process divides its cells into red/black groups, forming a checker board. In the first half of the iteration only red cells are update. Then halos are exchanges. Then the black cells are update, follwed by exchange of halos.

2. Applying the operator A (from $Ax = b$) using the stencil method.

The diffusion equation can be written in the form:

$$-D\left[\frac{\phi_{i+1,j} - 2\phi_{i,j} + 2\phi_{i+1,j}}{\delta^2} + \frac{\phi_{i,j-1} - 2\phi_{i,j} + 2\phi_{i,j+1}}{\gamma^2}\right] + \Sigma_a \phi_{i,j} = q_{ij} \tag{1}$$

where D is the diffusion factor, Σ_a - the macroscopic absorption cross-section, $q_{i,j}$ is the source strenght in the cell (x_i, y_j), $\phi_{i,j}$ is the scalar flux in the cell (x_i, y_j), and δ and γ are the cell sizes in x and y respectively. This produces a matrix with only 5 non-zero diagonals.

Therefore, it is not necessary to store the entire matrix, but only the way to update the flux value based on its neighbors, using the constants found in the equation. This means that at every iteration the following the next flux iterate of flux is obtained usin:

$$\phi_{i,j}^{(k+1)} = \phi_{i,j}^{(k)} + \frac{1}{s_{i,j}^{(c)}} \left(q_{i,j} - (s_{i,j}^{(b)} \phi_{i-1,j}^{(k)} + s_{i,j}^{(t)} \phi_{i+1,j}^{(k)} + s_{i,j}^{(l)} \phi_{i,j-1}^{(k)} + s_{i,j}^{(r)} \phi_{i,j-1}^{(k)}) \right) \tag{2}$$

where $s_{i,j}^{(c)} = \Sigma_a + 2D\left(\delta^{-2} + \gamma^{-2}\right)$, $s_{i,j}^{(l)} = s_{i,j}^{(r)} = -D\gamma^{-2}$, and $s_{i,j}^{(t)} = s_{i,j}^{(b)} = -D\delta^{-2}$

3 2. Numerical Experiments

Two major experiments are conducted to verify the correctness and scaling of the code:

1. Comparison against LUP methods and serial version from Previous Milestones.

A small system of $n = 12$ is used to compare the resulting flux from LUP, Serial PJ, Parallel PJ.

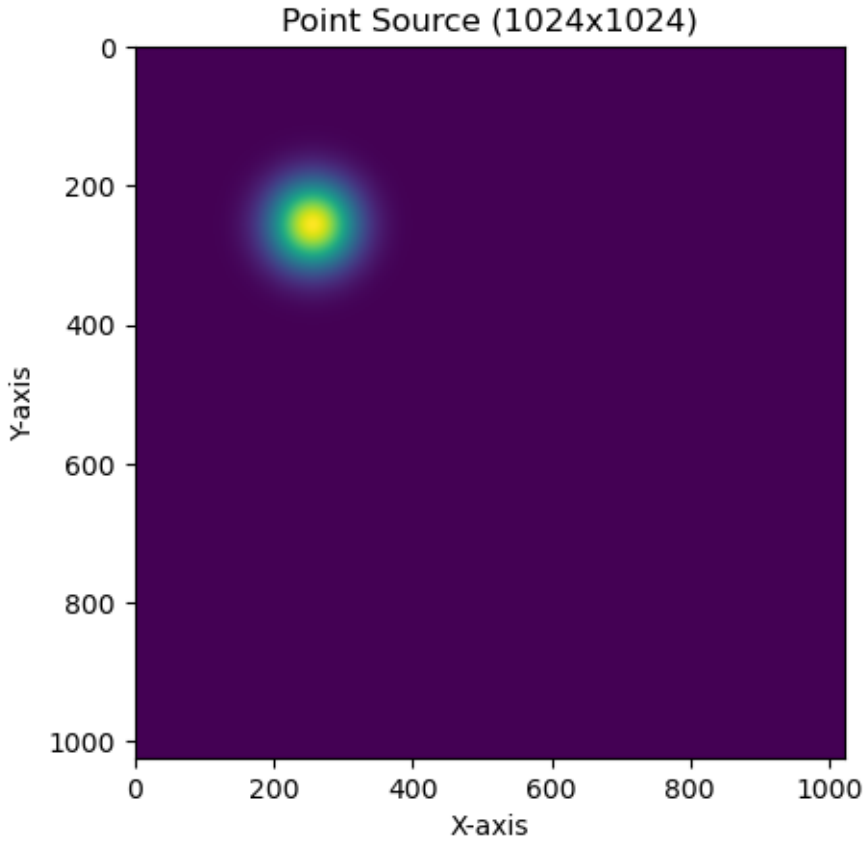
2. Timing measurements as a function of n - the size of the system, and P - number of processors used.

To complete the second experiment, we selected four system with sizes $n = \{128, 256, 512, 1024\}$, uniformly spaced grids and unit $D = 1$, and $\Sigma_a = 0.1$. The systems have a singular point source located in the top left corner of the rectangular region. The solutions, $\phi_{i,j}$ are shown above for each system. All iterative methods were allowd to run for maximum of $500,000$ iterations with tolerance of 10^{-5} and relaxation fractor 1.905 (for SOR).

[184]:
```python
from utils import Grid, System, Parameters, Inputs

grid_sizes = [128, 256, 512, 1024]


def generate_input_file(n: int) -> None:
    params = Parameters('gs', 1e-5, 500_000, 1.95)

    system = System.from_point_sources(
        Grid.build_square(10, n),
        locs=(2.5, 2.5),
        fwhms=1.0,
        D=1, S=0.1
    )

    inputs = Inputs(params, system)

    filepath = tests_dir / f'g{n}_{params.algorithm}.inp'
    inputs.to_txt(filepath)

    return inputs
```

```
inputs = []
for n in grid_sizes:
    inp = generate_input_file(n)
    inputs.append(inp)
```

[185]:
```
inp = inputs[-1]
img = plt.imshow(inp.system.source, cmap='viridis')  # Use 'viridis' colormap
plt.xlabel("X-axis")
plt.ylabel("Y-axis")

n = inp.system.grid.M
plt.title(f"Point Source ({n}x{n})")
plt.show()
```



4 3. Results

[186]:
```
def parse_output(filename: str) -> dict:
    data = {}
    with open(filename, 'r') as f:
        for line in f:
            if line.startswith('Algorithm'):
                data['algo'] = line.split(':')[-1].strip()

            if line.startswith('Non-'):
                data['n'] = int(line.split()[-1])

            if line.startswith('Processes'):
                data['p'] = int(line.split()[-1])

            if line.startswith('Converged'):
                data['conv'] = bool(line.split()[-1].title())

            if line.startswith('#Iterations:'):
                data['iters'] = int(line.split()[-1])

            if line.startswith('Iterative Error:'):
                data['iter_err'] = float(line.split()[-1])

            if line.startswith('Max'):
                data['res_err'] = float(line.split()[-1])

            if line.startswith('Execution'):
                data['time'] = float(line.split()[2])

    return data


def get_data():
    data = []
    for f in tests_dir.glob('*.out'):
        d = parse_output(f)

        suffix = f.suffixes[-2].split('_')
        d['m'] = int(suffix[-1]) if len(suffix) > 1 else 0
        data.append(d)
    return data
```

[187]:
```
data = get_data()
df = pd.DataFrame.from_dict(data)
df = df.dropna(axis=0)
dfs = df[(df.m == 4) | (df.p == 1)]
```

```
dfm = df[df.m != 4]
dfgm = dfm.groupby(['algo', 'n', 'p']).agg(['mean', 'std']).sort_index()
dfgs = dfs.groupby(['algo', 'n', 'p']).agg(['mean', 'std']).sort_index()
```

4.1 Verification

Verification according to the first experiment has been completed. The result flux and absolute residual are identifcal between Parallel and Serial PJ/GS/SOR code, providing the same relative error of ~1e5 wrt to LUP. Note the the number of iterations, iterative error and residual is the same for the same algorithm, i.e. number of processes does not affect the outcome, only the time.

4.2 Scaling

Below is the table with the results of the calculations. All systems converged, have the exact same number of iterations, relative error, and residual error, for each respective n, independent of p, which points that our implementation is synchronous (given the checker boarding ordering of the update).

[188]:
```
dfgs
```

[188]:

			res_err		conv		iters
			mean	std	mean	std	mean
algo	n	p					
Gauss-Seidel	128.0	1.0	0.000058	0.0	1.0	0.0	7299.0
		4.0	0.000058	0.0	1.0	0.0	7299.0
		16.0	0.000058	0.0	1.0	0.0	7299.0
	256.0	1.0	0.000228	0.0	1.0	0.0	22881.0
		4.0	0.000228	0.0	1.0	0.0	22881.0
		16.0	0.000228	0.0	1.0	0.0	22881.0
	512.0	4.0	0.000881	0.0	1.0	0.0	67282.0
		16.0	0.000881	0.0	1.0	0.0	67282.0
	1024.0	16.0	0.003257	0.0	1.0	0.0	179062.0
Point Jacobi	128.0	1.0	0.000058	0.0	1.0	0.0	13051.0
		4.0	0.000058	0.0	1.0	0.0	13051.0
		16.0	0.000058	0.0	1.0	0.0	13051.0
	256.0	1.0	0.000225	0.0	1.0	0.0	39703.0
		4.0	0.000225	0.0	1.0	0.0	39703.0
		16.0	0.000225	0.0	1.0	0.0	39703.0
	512.0	4.0	0.000854	0.0	1.0	0.0	111497.0
		16.0	0.000854	0.0	1.0	0.0	111497.0
	1024.0	16.0	0.003086	0.0	1.0	0.0	278576.0
Successive Over Relaxation	128.0	1.0	0.000027	0.0	1.0	0.0	235.0
		4.0	0.000027	0.0	1.0	0.0	235.0
		16.0	0.000027	0.0	1.0	0.0	235.0
	256.0	1.0	0.000116	0.0	1.0	0.0	931.0
		4.0	0.000116	0.0	1.0	0.0	931.0
		16.0	0.000116	0.0	1.0	0.0	931.0
	512.0	4.0	0.000468	0.0	1.0	0.0	3316.0

		16.0	0.000468	0.0	1.0	0.0	3316.0
	1024.0	16.0	0.001866	0.0	1.0	0.0	10908.0

				iter_err		time	\
			std	mean	std	mean	
algo	n	p					
Gauss-Seidel	128.0	1.0	0.0	0.000010	0.0	1.791585	
		4.0	0.0	0.000010	0.0	0.539575	
		16.0	0.0	0.000010	0.0	0.224487	
	256.0	1.0	0.0	0.000010	0.0	23.043811	
		4.0	0.0	0.000010	0.0	6.123723	
		16.0	0.0	0.000010	0.0	1.820115	
	512.0	4.0	0.0	0.000010	0.0	10.261133	
		16.0	0.0	0.000010	0.0	17.976968	
	1024.0	16.0	0.0	0.000010	0.0	10.501886	
Point Jacobi	128.0	1.0	0.0	0.000010	0.0	2.822069	
		4.0	0.0	0.000010	0.0	0.816221	
		16.0	0.0	0.000010	0.0	0.314004	
	256.0	1.0	0.0	0.000010	0.0	34.157059	
		4.0	0.0	0.000010	0.0	9.121504	
		16.0	0.0	0.000010	0.0	2.598054	
	512.0	4.0	0.0	0.000010	0.0	40.104337	
		16.0	0.0	0.000010	0.0	25.612413	
	1024.0	16.0	0.0	0.000010	0.0	21.631628	
Successive Over Relaxation	128.0	1.0	0.0	0.000009	0.0	0.058958	
		4.0	0.0	0.000009	0.0	0.019563	
		16.0	0.0	0.000009	0.0	0.011836	
	256.0	1.0	0.0	0.000010	0.0	0.941494	
		4.0	0.0	0.000010	0.0	0.254889	
		16.0	0.0	0.000010	0.0	0.079335	
	512.0	4.0	0.0	0.000010	0.0	3.485821	
		16.0	0.0	0.000010	0.0	0.908005	
	1024.0	16.0	0.0	0.000010	0.0	11.745983	

				m		
			std	mean	std	
algo	n	p				
Gauss-Seidel	128.0	1.0	0.001951	0.0	0.0	
		4.0	0.016920	4.0	0.0	
		16.0	0.014719	4.0	0.0	
	256.0	1.0	0.097063	0.0	0.0	
		4.0	0.117954	4.0	0.0	
		16.0	0.027911	4.0	0.0	
	512.0	4.0	0.457002	4.0	0.0	
		16.0	0.424022	4.0	0.0	
	1024.0	16.0	3.210703	4.0	0.0	
Point Jacobi	128.0	1.0	0.004857	0.0	0.0	

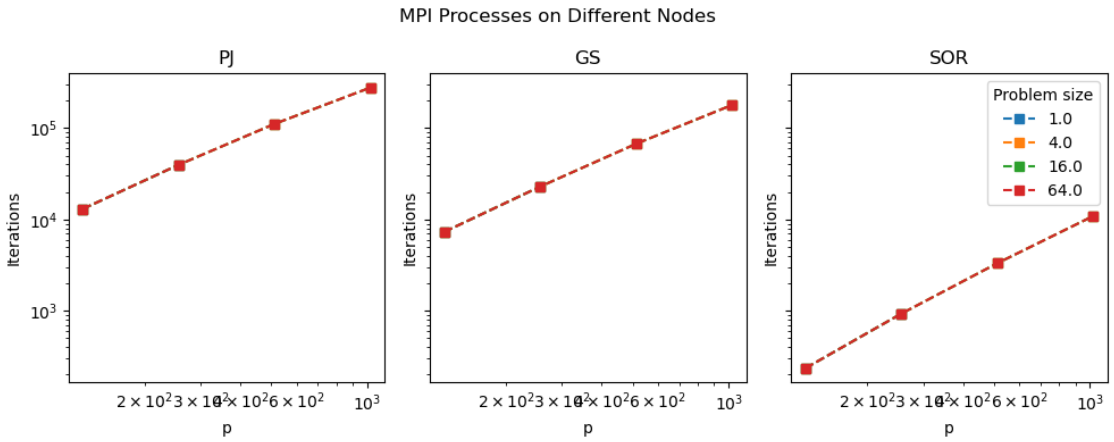
```
                                       4.0   0.017315  4.0  0.0
                                       16.0  0.018054  4.0  0.0
                              256.0    1.0   0.113633  0.0  0.0
                                       4.0   0.047739  4.0  0.0
                                       16.0  0.022171  4.0  0.0
                              512.0    4.0   0.471478  4.0  0.0
                                       16.0  0.555254  4.0  0.0
                              1024.0   16.0  1.531655  4.0  0.0
Successive Over Relaxation    128.0    1.0   0.000274  0.0  0.0
                                       4.0   0.001308  4.0  0.0
                                       16.0  0.003980  4.0  0.0
                              256.0    1.0   0.004192  0.0  0.0
                                       4.0   0.005943  4.0  0.0
                                       16.0  0.005487  4.0  0.0
                              512.0    4.0   0.022216  4.0  0.0
                                       16.0  0.015664  4.0  0.0
                              1024.0   16.0  0.192143  4.0  0.0
```

4.3 Iterations

Below is the number of iterations required for each algorithm to complate. SOR is a clear winner.

[189]:
```
fig, axes = plt.subplots(1, 3, layout='tight', figsize=(10, 4), sharex=True,␣
 ↪sharey=True)

for (algo, n), dfa in dfgm.reset_index().groupby(['algo', 'p']):
    if algo.lower()[0] == 'p':
        ax = axes[0]

    if algo.lower()[0] == 'g':
        ax = axes[1]

    if algo.lower()[0] == 's':
        ax = axes[2]

    ax.loglog(dfa['n'], dfa[('iters', 'mean')], marker='s', ls='--', label=(n␣
 ↪if algo[0].lower() == 's' else None))

axes[2].legend(title='Problem size')
for ax in axes:
    ax.set_xlabel('p')
    ax.set_ylabel('Iterations')

axes[0].set_title('PJ')
axes[1].set_title('GS')
axes[2].set_title('SOR')
plt.suptitle('MPI Processes on Different Nodes')
```

```
plt.show()
```



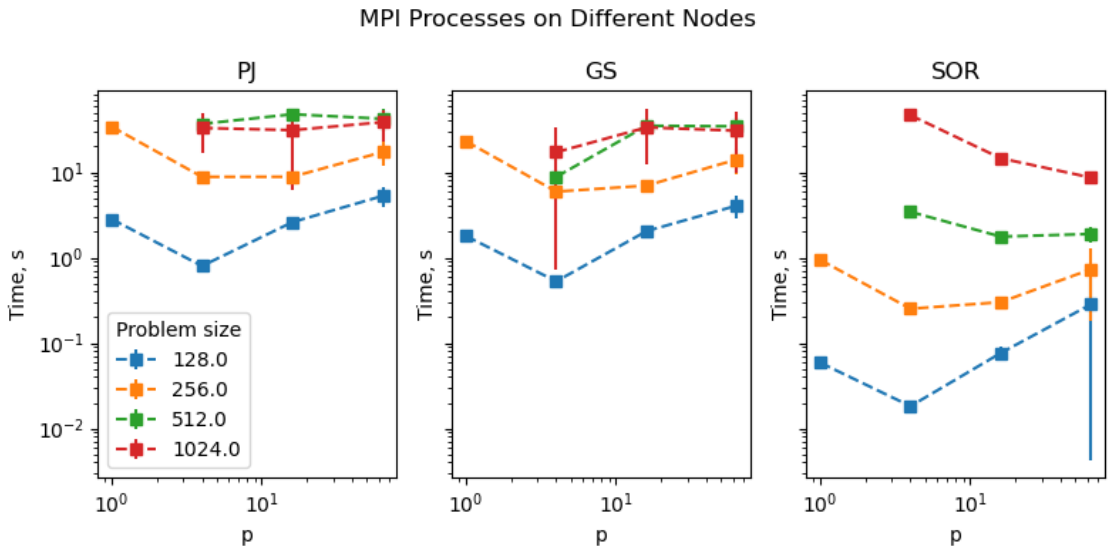
4.4 Execution time

Below is the plot of execution time as a function of number of processors used to solve the problem. It can be seen that generally that larger problems result in longer execution time. There is a general downward trend (especially for single machines) with increase of number of processors. However, there are some exceptions. It should be noted that there's a lot of variability in time measuremnts (error bars given in one +- 1 STD), which can be explained by the local state of the machine and network at the moment of the measurement.

4.4.1 Multiple Machines

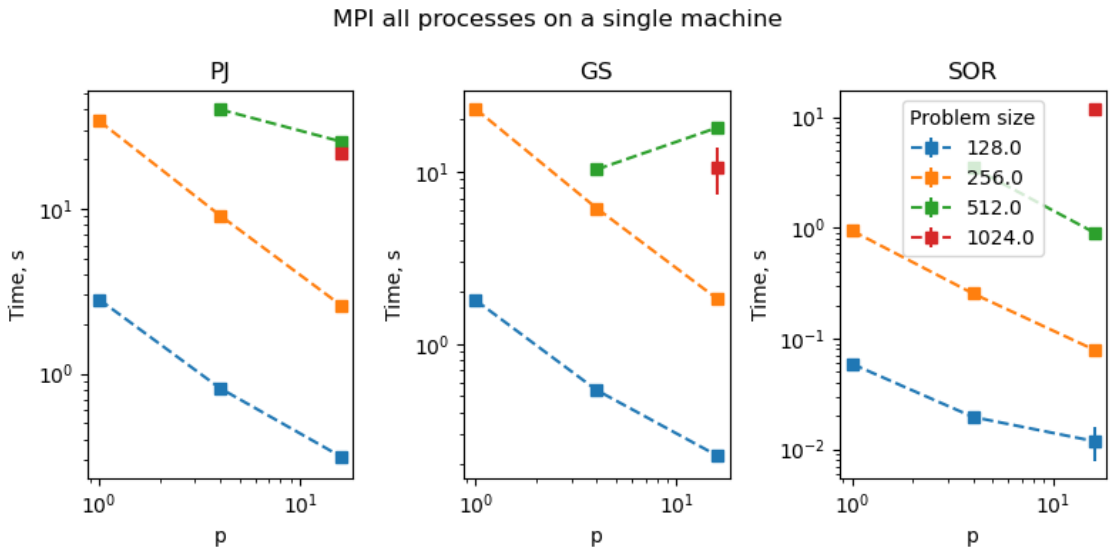
[190]:
```
fig, axes = plt.subplots(1, 3, layout='tight', figsize=(8, 4), sharex=True,␣
↪sharey=True)

for (algo, n), dfa in dfgm.reset_index().groupby(['algo', 'n']):
    if algo.lower()[0] == 'p':
        ax = axes[0]

    if algo.lower()[0] == 'g':
        ax = axes[1]

    if algo.lower()[0] == 's':
        ax = axes[2]

    ax.errorbar(dfa['p'], dfa[('time', 'mean')], yerr=dfa[('time', 'std')],␣
↪marker='s', ls='--', label=(n if algo[0].lower() == 'p' else None))

axes[0].legend(title='Problem size')
for ax in axes:
```

```
    ax.set_xlabel('p')
    ax.set_ylabel('Time, s')
    ax.set_xscale('log')
    ax.set_yscale('log')

axes[0].set_title('PJ')
axes[1].set_title('GS')
axes[2].set_title('SOR')
plt.suptitle('MPI Processes on Different Nodes')
plt.show()
```



[191]:
```
fig, axes = plt.subplots(1, 3, layout='tight', figsize=(8, 4))

for (algo, n), dfa in dfgs.reset_index().groupby(['algo', 'n']):
    if algo.lower()[0] == 'p':
        ax = axes[0]

    if algo.lower()[0] == 'g':
        ax = axes[1]

    if algo.lower()[0] == 's':
        ax = axes[2]

    ax.errorbar(dfa['p'], dfa[('time', 'mean')], yerr=dfa[('time', 'std')],␣
 ↪marker='s', ls='--', label=(n if algo[0].lower() == 's' else None))

axes[2].legend(title='Problem size')
for ax in axes:
```

```
    ax.set_xlabel('p')
    ax.set_ylabel('Time, s')
    ax.set_xscale('log')
    ax.set_yscale('log')

axes[0].set_title('PJ')
axes[1].set_title('GS')
axes[2].set_title('SOR')
plt.suptitle('MPI all processes on a single machine')
plt.show()
```



5 4. Discussion

The implementation has been verified to be both correct and synchronous. The "general" reduction of executation time is visiable by in the graph above, however, it is obscured by the large variability of the associated with each data point. Another explanation for the "upticks" can be explained by the rising "cost" of maintaining communication between large number of processor. The update of halo gets more expensive with the growing number of processors. And it becomes expecially prominent when the block size becomes comparable to the halo size. This observation points to an existence of optimal number of processors for the given problem size, aka "less" is sometimes "more".

Finally, for small problems massive parallelization is hinderance, since cost of communication becomes comparable with the time required to solve the problem.

6 5. Conclusion

We have implemented solution of finite difference methods using parallel Point-Jacobi, GS, and SOR algorithms. Using the stencil and checker-board techniques, we were able to achieve synchronous parallel implementation, which numerically practically indistinguishible from the serial code. We have observed reduction of execution time with the number of processors used. However, this general trend is obscured by the inefficiency of process-to-process communication and the state of compute machine and network at the moment of computation.