Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High-Performance Computing Lab for CSE **2024**

Student: Timo Schwab Discussed with: Lily Watanabe

Solution for Project 6

Due date: Monday 27 May 2024, 23:59 (midnight).

HPC Lab for CSE 2024 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to Moodle (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. Graph Partitioning with Julia: Load balancing for HPC [50 points]

1.1. Inertial Bisection

We compute the center of mass and construct the matrix $\underline{\underline{M}}$ the following way:

```
COM = mean(coords, dims=1)
# construct matrix
for i in 1:N
    S_xx += (coords[i,1] - COM[1])^2
    S_xy += (coords[i,1] - COM[1]) * (coords[i,2] - COM[2])
    S_yy += (coords[i,2] - COM[2])^2
end
M = [ S_xx S_xy;
      S_xy S_yy ]
```

Then the eigs function is used to compute the eigenvalue and associated eigenvector of smallest magnitude. Then the previously given partition method is used to bisect the graph along the eigenvector of smallest magnitude.

```
# compute smallest eigenvalue and associated eigenvector
eigenvalue_min, eigenvector_min = eigs(M, nev=1, which=:SM)
# partition along smallest eigenvector
p1, p2 = partition(coords, eigenvector_min)
```

1.2. Spectral Bisection

When computing the fiedler vectors on EULER Jupyterhub, an error occured when using Arpack version 0.5.4. Thus, usage of Arpack 0.5.3 had to be manually specified in all main scripts via

```
Pkg.add(Pkg.PackageSpec(name="Arpack", version="0.5.3"))
```

The Laplacian matrix is constructed using

```
column_sums = sum(A, dims=1)
L = Diagonal(column_sums[1, :]) .- A
```

Then the Fiedler vector is computed via Arpack's `eigs` function, if `maxiter` is not specified, the recursive bisection using the spectral method will not converge in all cases.

```
maxiter = 10000
eigenvals, eigenvecs = eigs(L, nev=2, which=:SR, maxiter=maxiter)
# Fiedler vector is eigenvector associated with the second smallest eigenvalue
fiedler_vec = eigenvecs[:, 2]
```

Now we partition the graph into values smaller or larger equal the median value of the fiedler vector

```
epsilon = median(fiedler_vec)
p = ones(Int, length(fiedler_vec))
for i in 1:length(fiedler_vec)
    if fiedler_vec[i] < epsilon
        p[i] = 1
    else
        p[i] = 2
    end
end
```

1.3. Discussion

Metis outperforms the methods considered in almost all cases. However, partitionings of similar qualtiy can be achieved in some cases. For the non-recursive methods we look at four cases.

1. "mesh1e1"; The linear cuts (1, 2, 3) are a good approximation to the cut generated by Metis (4), hence we get bisections of similar quality.

2. "barth4"; The cut generated by Metis (6) is circular and covers only a tiny part of the mesh. Clearly, this cannot be approximated by any of the linear cuts, the inertial bisection (5) delivers a particularly bad result here.

Mesh	Coordinate	Metis v.5.1.0	Inertial	Spectral
mesh1e1	18.0	17.0	20.0	18.0
mesh2e1	37.0	34.0	47.0	35.0
mesh3e1	19.0	18.0	22.0	18.0
airfoil1	94.0	73.0	93.0	132.0
netz4504_dual	25.0	20.0	27.0	23.0
stufe	16.0	17.0	16.0	16.0
3elt	172.0	90.0	257.0	117.0
barth4	206.0	100.0	208.0	127.0
ukerbe1	32.0	30.0	28.0	28.0
crack	353.0	200.0	384.0	233.0

Table 1: Performance comparison of graph bisection on different meshes vs. Metis

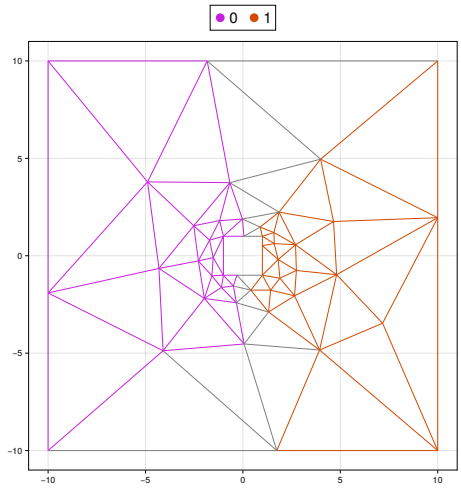

Figure 1: Coord. bisection of mesh1e1

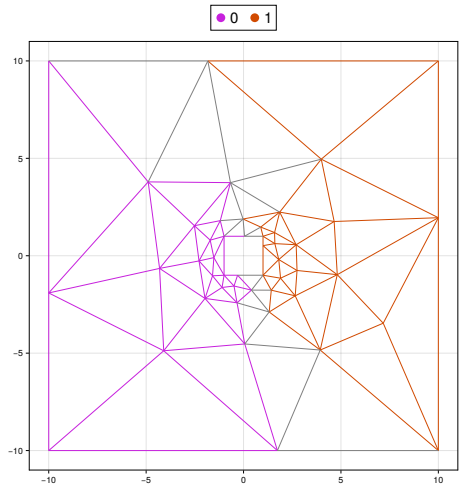

Figure 2: Inertial bisection of mesh1e1

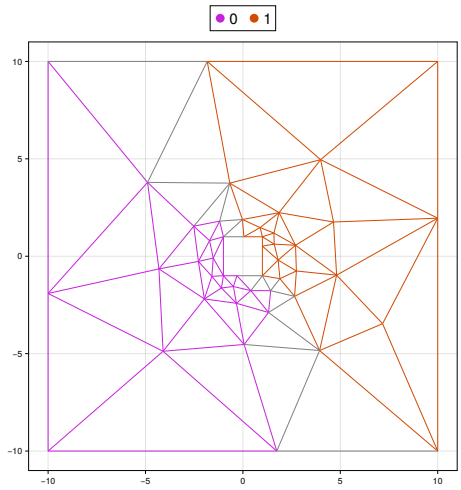

Figure 3: Spectral bisection of mesh1e1

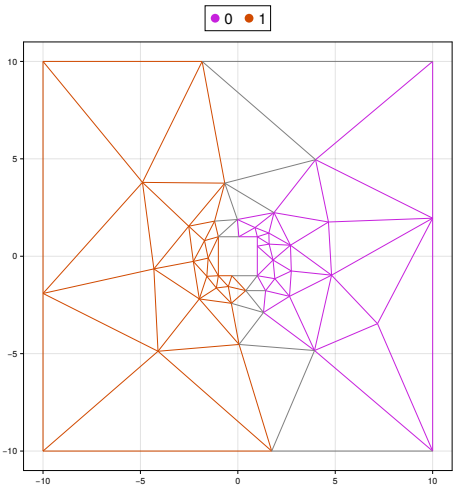

Figure 4: Result computed by Metis

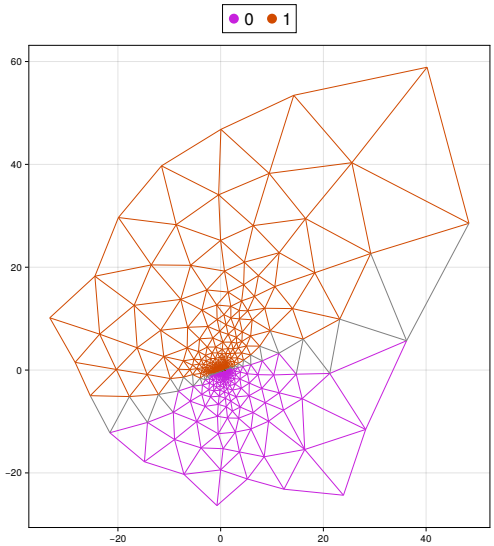

Figure 5: Inertial Bisection of 3elt

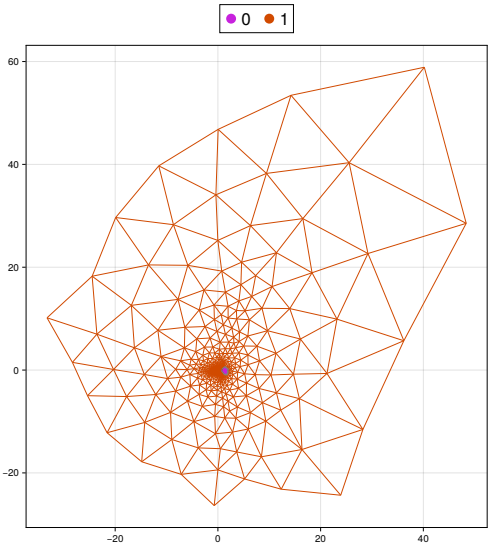

Figure 6: Result computed by Metis

3. "airfoil"; The result computed by Metis (8) uses empty spaces of the airfoil mesh to decrease the amount of edges on the cut. The spectral bisection (7) created a circular cut not using any of the empty spaces, this cut is inferior even to some of the linear cuts computed.

4. "ukerbe1"; This mesh is the only case the Spectral (9) and Inertial Methods beat the Metis result (10). Where Metis computes a circular cut in the center of the mesh, the spectral method computes a mush simpler, almost linear cut through the middle.

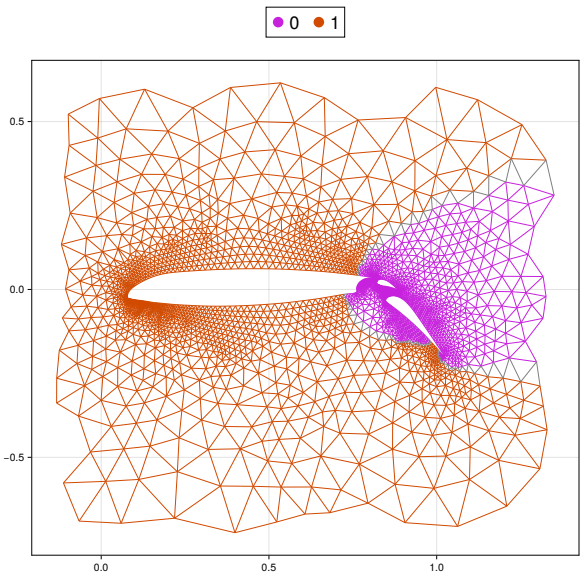

Figure 7: Spectral Bisection of airfoil

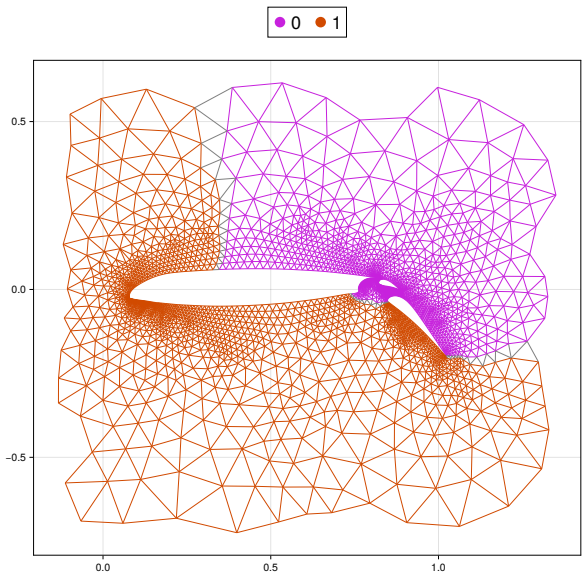

Figure 8: Result computed by Metis

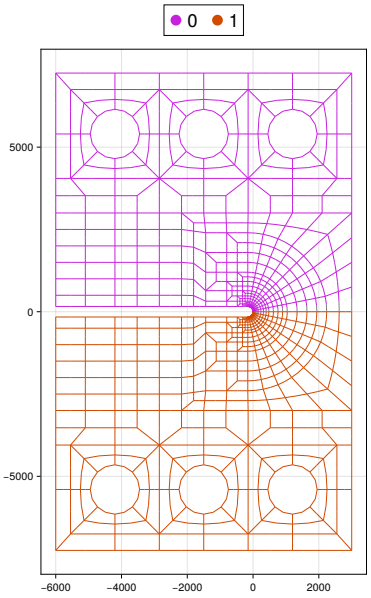

Figure 9: Spectral Bisection of ukerbe1

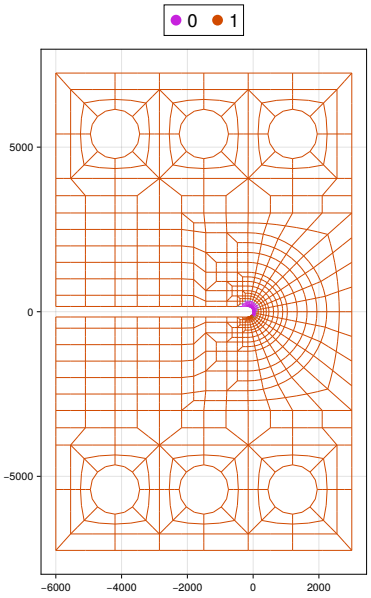

Figure 10: Result computed by Metis

1.4. Recursive Bisection

We terminate the recursion if a minimum point threshold is reached or the number of levels is smaller than one ($2^0 = 1$). Within the recursion we distinguish between two different types of method calls; this is because the spectral bisection does not require the `coords` argument.

```
if !isempty(coords)
    p = eval(Symbol(method))(A, coords)
    idx1 = findall(x -> x == 1, p)
    idx2 = findall(x -> x == 2, p)
    coords1 = coords[idx1, :]
    coords2 = coords[idx2, :]
else
    # equal but set coords = zero
end
```

To retrieve the partitioned subset we need to also partition `A` for later recursive calls.

```
vn1 = vn[idx1]
vn2 = vn[idx2]
A1 = A[idx1, idx1]
A2 = A[idx2, idx2]
```

Then, if the end of recursion is not yet reached, we continue by calling the method recursively on the subsets partitioned in the respective recursive call.

```
p1 = rec_bisection(method, levels, A1, coords1, vn1)
p2 = rec_bisection(method, levels, A2, coords2, vn2)
```

1.5. Discussion of Recursive Bisection Results

For improved readability, the tables are separated into the 8- and 16- way partitioning results.

Mesh	Coordinate	Metis (kway)	Metis (rec)	Inertial	Spectral
mesh3e1	75.0	71.0	74.0	150.0	74.0
airfoil1	516.0	321.0	324.0	672.0	397.0
netz4504_dual	127.0	99.0	101.0	165.0	111.0
stufe	123.0	113.0	110.0	324.0	128.0
3elt	733.0	383.0	402.0	814.0	469.0
barth4	875.0	419.0	409.0	977.0	550.0
ukerbe1	225.0	119.0	140.0	340.0	126.0
crack	1344.0	776.0	736.0	1351.0	883.0

Table 2: Performance comparison of different meshes using various methods for 8-way partitioning

Mesh	Coordinate	Metis (kway)	Metis (rec)	Inertial	Spectral
mesh3e1	118.0	217.0	112.0	300.0	121.0
airfoil1	819.0	545.0	566.0	1080.0	630.0
netz4504_dual	198.0	160.0	159.0	268.0	182.0
stufe	228.0	193.0	196.0	607.0	243.0
3elt	1168.0	636.0	645.0	1230.0	752.0
barth4	1306.0	683.0	716.0	1494.0	840.0
ukerbe1	374.0	230.0	237.0	499.0	238.0
crack	1861.0	1186.0	1238.0	1884.0	1419.0

Table 3: Performance comparison of different meshes using various methods 16-way partitioning

The Inertial bisection method does not seem suitable for recursive partitionings as it performs even worse than the coordinate bisection in all cases. Out of the methods implemented, Spectral bisection produces the most stable results and manages to beat recursive metis in the 8-way partitioning (11 and 12) and kway metis in 16-way partitioning (13 and 14).

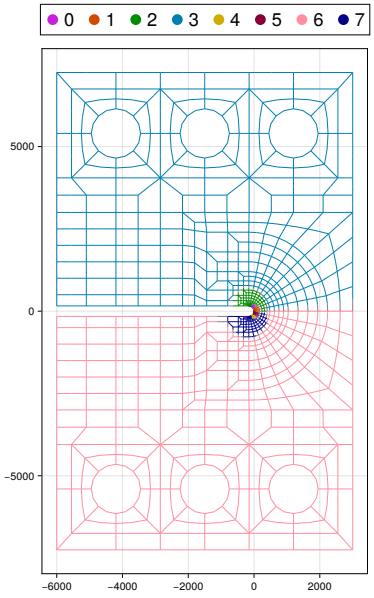

Figure 11: 8-way Spectral Bisection of ukerbe1

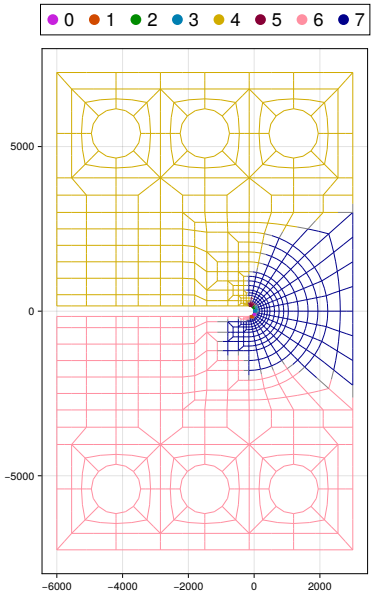

Figure 12: Result computed by 8-way Recursive Metis

The Spectral bisection of mesh3e1 (13) generates multiple isolated, circular clusters while the 16-way Metis (14) produces much more chaotic shapes. With a score of 121 (spectral) vs. 217 (kway), the cut generated by metis is significantly worse for such a regular mesh topology.

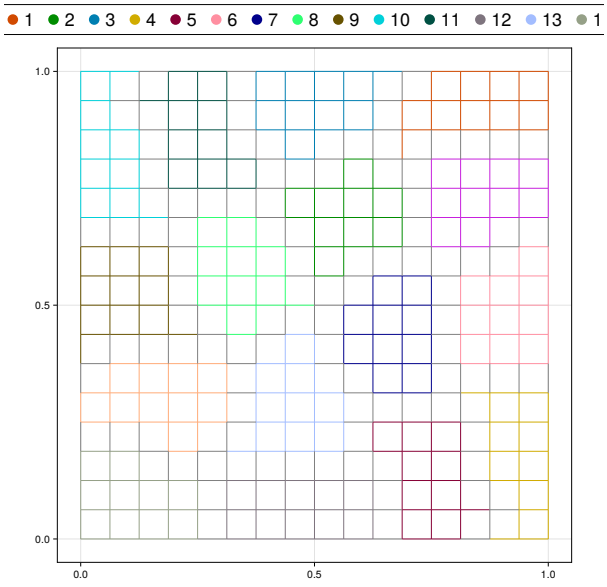

Figure 13: 16-way Spectral Bisection of mesh3e1

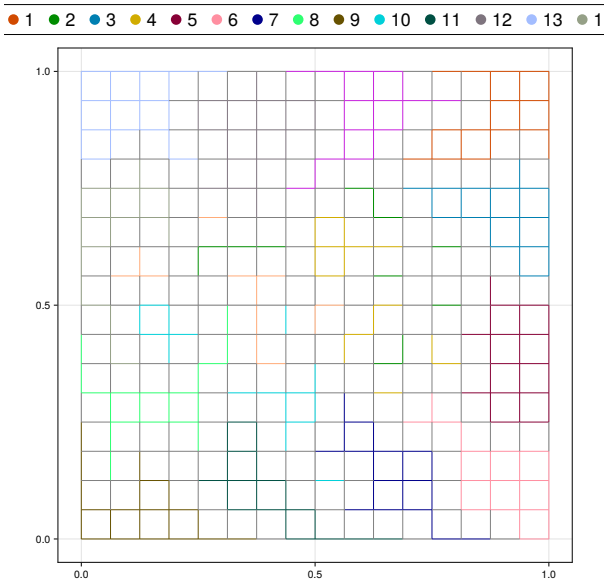

Figure 14: Result computed by 16-way Metis (kway)

2. HPC software frameworks [50 points]

The problem considered is solving the Poisson Equation using second order finite differences. The Poisson Equation is given by

$$-\Delta u = f \quad \text{in} \quad \Omega \tag{1}$$
$$u = 0 \quad \text{in} \quad \partial\Omega \tag{2}$$

where the second equation specifies zero Dirichlet boundary conditions and f is a constant source function. The computational domain Ω is chosen to be the unit square

$$\Omega = \{x = (x_1, x_2) \,|\, 0 < x_1, x_2 < 1\}$$

The chosen discretization is second-order centered finite-differences, which applied to the Poisson Equation yields

$$-\Delta u = -\left(\frac{\partial^2 u(x_1,x_2)}{\partial x_1^2} + \frac{\partial^2 u(x_1,x_2)}{\partial x_2^2}\right) = f \tag{3}$$
$$= -\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}\right) = f \tag{4}$$
$$= \frac{4u_{i,j}}{h^2} - \frac{u_{i+1,j}}{h^2} - \frac{u_{i-1,j}}{h^2} - \frac{u_{i,j+1}}{h^2} - \frac{u_{i,j-1}}{h^2} = f \tag{5}$$

where h is the meshwidth and $f = 20$. This yields a $m^2 \times m^2$ Linear System of equations

$$\underline{\underline{A}} \cdot \underline{u} = \underline{b}$$

$\underline{\underline{A}}$ is a sparse matrix with entries $\frac{4}{h^2}$ on its diagonal and $\frac{-1}{h^2}$ on four offdiagonals corresponding to $u_{i+1,j}$, $u_{i-1,j}$, $u_{i,j+1}$, $u_{i,j-1}$ respectively. $\underline{u}$ is the solution vector

$$\underline{\underline{A}} = \frac{1}{h^2}\left(\begin{array}{ccc|ccc|ccc} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ \hline -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ \hline 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{array}\right), \quad \underline{u} = \begin{pmatrix} u_{11} \\ u_{12} \\ u_{13} \\ u_{21} \\ u_{22} \\ u_{23} \\ u_{31} \\ u_{32} \\ u_{33} \end{pmatrix}$$

$\underline{b}$ enforces the boundary conditions and the constant source function f

$$b_{ij} = \begin{cases} 0, & \text{if the corresponding } u_{ij} \in \partial\Omega, \\ 20, & \text{if the corresponding } u_{ij} \in \Omega. \end{cases}$$

2.1. Implementation in PETSc

The matrix $\underline{\underline{A}}$ and vector $\underline{b}$ is constructed in parallel (distributed) using PETSc's DMDA routines. `PETSC_DECIDE` indicates to PETSc that the distribution across ranks is the chosen to be PETSc's default. As we do not simulate under periodic boundary conditions or explicitly manage ghost cells, the boundary is simply `DM_BOUNDARY_NONE`.

```
PetscCall(DMDACreate2d(PETSC_COMM_WORLD, DM_BOUNDARY_NONE, DM_BOUNDARY_NONE,
                           stype, m, m, PETSC_DECIDE, PETSC_DECIDE, 1, 1, NULL,
                              NULL, &da ));
PetscCall(DMSetFromOptions(da));
PetscCall(DMSetUp(da));

// create matrix A and vectors b, u
PetscCall(DMCreateMatrix(da, &A));
PetscCall(DMCreateGlobalVector(da, &b));
PetscCall(VecDuplicate(b, &u));
```

The values are set using the star stencil operation provided by PETSc. Each rank only initializes values in the range assigned to the rank, obtained via `DMDAGetCorners`.

```
PetscCall(DMDAGetCorners(da, &xmin, &ymin, NULL, &xcount, &ycount, NULL));
for (PetscInt i = xmin; i < xmin + xcount; ++i) {
    for (PetscInt j = ymin; j < ymin + ycount; ++j) {
        row.i = i; row.j = j;
        PetscInt row_index = j * m + i;
        // boundary points have to be zero
        if (i==0 || j==0 || i==m-1 || j==m-1) {
            MatSetValuesStencil(A, 1, &row, 1, &row, &diag, INSERT_VALUES);
            VecSetValue(b, row_index, 0.0, INSERT_VALUES);
        } else {
            // set matrix from stencil
            v[0] = offdiag; col[0].i = i-1; col[0].j = j;
            v[1] = offdiag; col[1].i = i;   col[1].j = j-1;
            v[2] = diag;    col[2].i = i;   col[2].j = j;
            v[3] = offdiag; col[3].i = i;   col[3].j = j+1;
            v[4] = offdiag; col[4].i = i+1; col[4].j = j;
            MatSetValuesStencil(A, 1, &row, 5, col, v, INSERT_VALUES);
            // constant source
            VecSetValue(b, row_index, f, INSERT_VALUES);
        }
    }
}
```

For an efficient implementation, PETSC was given additional information about the structure of the matrix. The solver and preconditioner types were chosen empirically, Preconditioned Conjugate Gradient (KSPCG) was used in combination with Geometric Algebraic Mutigrid (PCGAMG).

```
MatSetOption(A, MAT_SYMMETRIC, PETSC_TRUE);
KSPCreate(PETSC_COMM_WORLD, &ksp);  // create ksp context
KSPSetType(ksp, KSPCG);             // set ksp to Preconditioned CG
PCSetType(pc, PCGAMG);              // set pc to Algebraic Multigrid
// ...
KSPSolve(ksp, b, u);                // solve LSE
```

2.2. Results

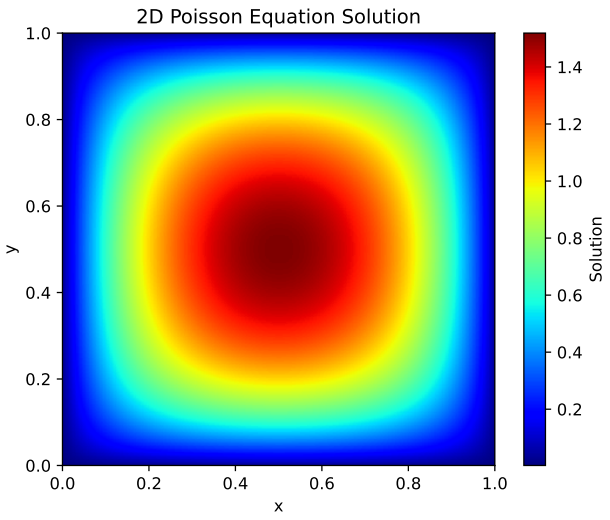

Figure 15: Solution for a gridsize of 1024×1024

The visualization of the calculated solution to the Poisson Equation can be seen in 15. The boundary values in the numerically computed result are zero, in agreement with the expected solution due to the chosen Dirichlet boundary conditions. There are no visible discontinuities and the solution seems very smooth, also as expected given the Poisson Equation is a elliptic partial differential equation, where characteristic curves would be a clear sign of error. Finally, comparing against other visualizations of solutions to the Poisson Equation with constant source function on the unit square it can be concluded the solver successfully computes a correct numerical approximation. In the benchmarks, the method outputting the solution values is commented out, to reproduce the plot this would have to be changed back.

2.2.1. Single Node Performance

Strong Scaling tests were conducted on EULER IX, a CPU (AMD EPYC_9654) based HPC Cluster with 48 ranks on the same node. The processors were configured with 2 Gigabytes of RAM due to the large memory requirements of the computational grid and representative matrix.

To reproduce the scaling results, two shell scripts are provided in the `scritps/` directory of the project. The shell script ending on `_1.sh` refers to the single node test, whereas the script ending on `_64.sh` refers to the distributed test. Problem size has to be adjusted manually in the script.

The scaling plots are generated using a python script `tripleplot.py`, located in the `strong/` directory. It plots the results for a given scaling run of arbitrary core counts and repetitions, but requires the input/output filename to be specified manually in the script.

The method responsible for generating a solution file that can be turned into the visualization 15 is commented out. If used, the poisson solver will generate a file `solution.txt` that can be turned into 15 by the `plot_poisson.py` script provided in the main `poisson_petsc` directory.

Performance was measured after three warmup runs, the median of 5 runs is reported in the plots. Per run, three time measurements are taken, reported is the maximum time of all ranks (i.e. the time it took for the slowest rank to complete the methods) for each one of the measurements taken.

- "Execution Time" refers to the total time taken by the program between MPI Initialization and Cleanup (excluding the MPI_Reduce required to gather the measurement results)
- "Solver Time" refers to the time spent in the `KSPSolve` function
- "Setup Time" refers to the time spent assembling the Matrix $\underline{\underline{A}}$ and Vector $\underline{b}$

The EULER IX Nodes were chosen due to the instability of some methods at larger rank counts (> 24) on the EULER VII architecture. Performance on EULER VII nodes differed by a factor of 10 between consecutive runs of the same problem size on the same node with the same rank count.

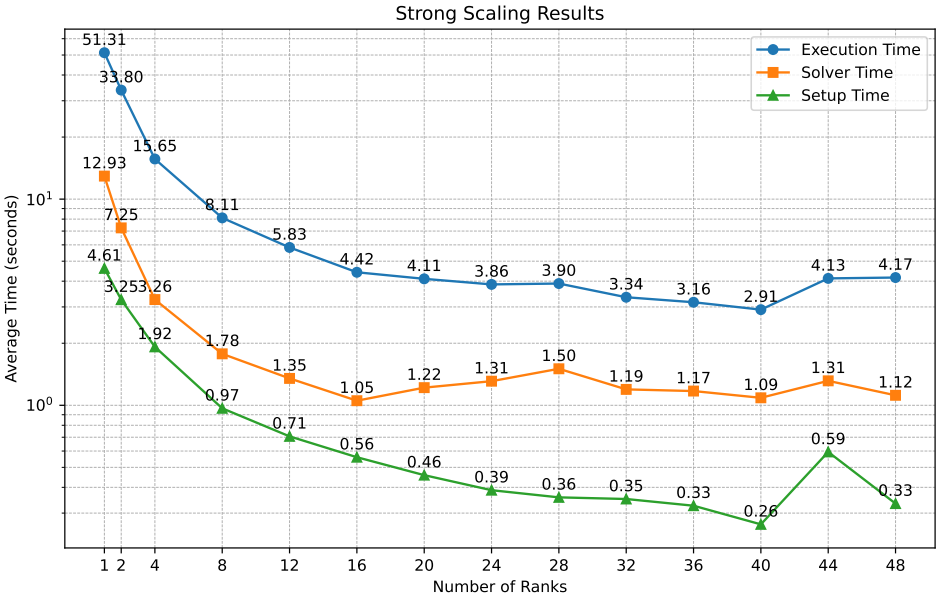

Figure 16: 4096 × 4096; computed on a single node of the EULER IX cluster with up to 48 cores of the AMD EPYC_9654 CPU (2.4 GHz nominal, 3.7 GHz peak) and 96 GB of RAM

For the matrix assembly methond we can see a stable decline in execution time for up to 32 (Fig. 16) and 40 (Fig. 17) CPU cores, interestingly the same cannot be said about the solver. Here, performance does not seem to improve in both runs after 16 MPI Ranks. Because Solver time dominates the assembly time, one could argue that the choice of 16 ranks presents the best tradeoff between performance and parallel efficiency. On more than 36 ranks, the method did not stably perform and is thus advised against. Although this is likely due to the increasing complexity of some MPI synchronization routines used in the petsc methods, implementation errors cannot be definitively ruled out.

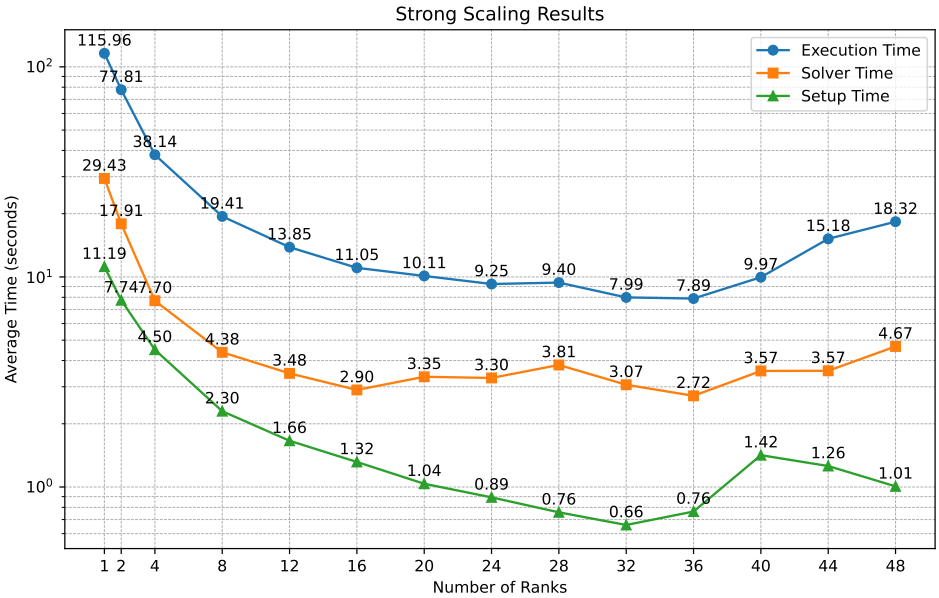

Figure 17: 6144 × 6144; computed on a single node of the EULER IX cluster with up to 48 cores of the AMD EPYC_9654 CPU (2.4 GHz nominal, 3.7 GHz peak) and 96 GB of RAM

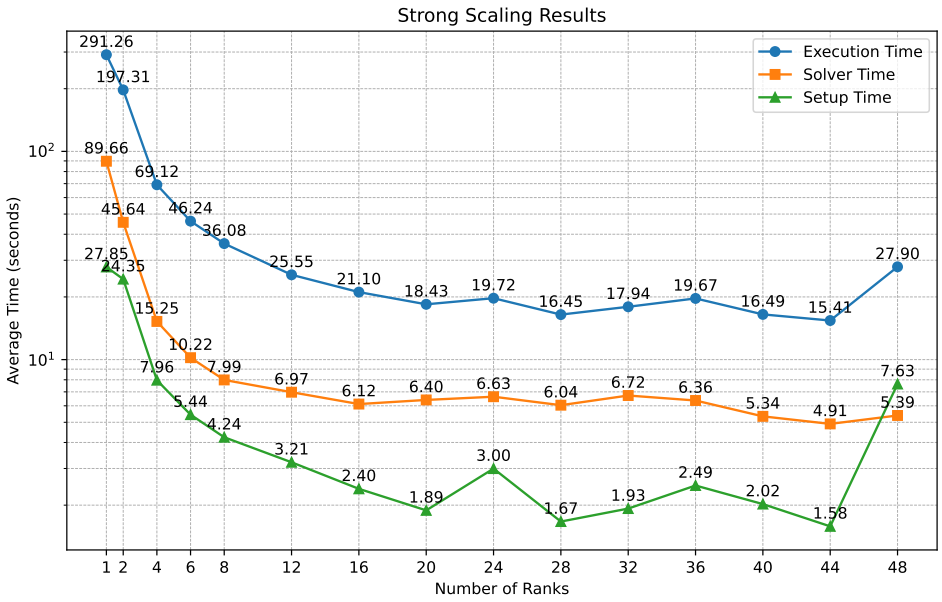

Figure 18: 8192×8192; computed on a single node of the EULER IX cluster with up to 48 cores of the AMD EPYC_9654 CPU (2.4 GHz nominal, 3.7 GHz peak) and 96 GB of RAM

Inconsistencies in performance can again be observed for larger rank counts. Although now the time improvements persist longer to up to 44 ranks in both 18 and 19. Solver time does, again, not show significant improvement after 16 ranks for the 8192×8192 grid, only the 10000×10000 grid is big enough for it so scale continuously for up to 28 ranks. It seems to be the setup time that causes some of the instabilities, although a thorough investigation of all functions executed would be required to give conclusive results.

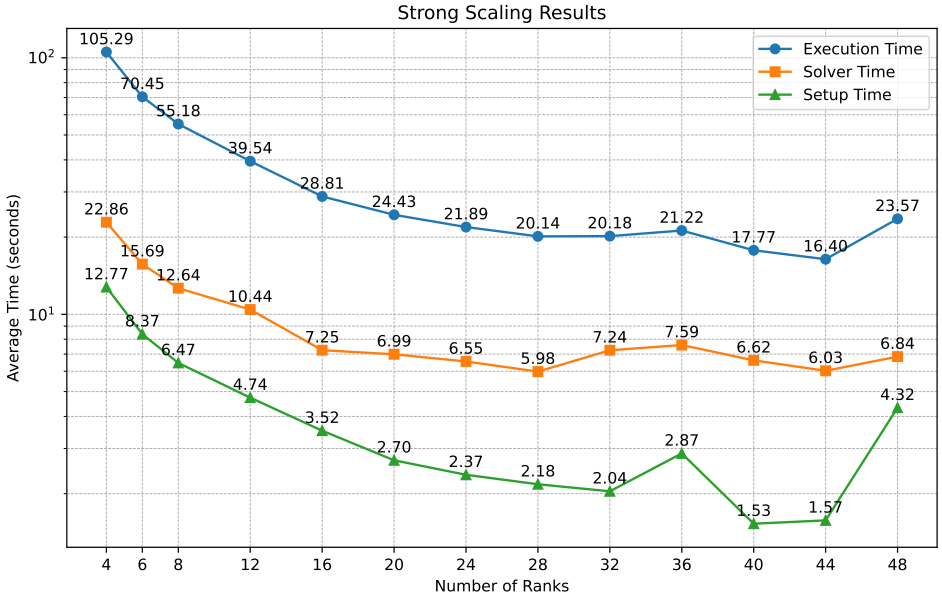

Figure 19: 10000×10000; computed on a single node of the EULER IX cluster with up to 48 cores of the AMD EPYC_9654 CPU (2.4 GHz nominal, 3.7 GHz peak) and 96 GB of RAM

2.3. Multi-Node Performance

To test the scalability of the implementation including node-to-node communication, strong scaling tests were conducted on up to 32 different nodes. In contrast to the single node tests, these ran on the EULER VII Cluster, a CPU based supercomputer with two AMD EPYC_7763 (2.45 GHz nominal, 3.5 GHz peak) processors on each node. For the distributed test, 4GB on each rank were requested (2 processors per rank with 2 GB RAM each), due to the lack of shared memory parallelism the second CPU on each rank should idle and not impact the scaling results compared to a single CPU run. It only serves the purpose of allowing higher memory-per-rank requests from the clusters slurm job managemet system by circumventing the big memory queue. Some timing data still had to be left out due to insufficient memory causing bad allocations (i.e. segfaults).

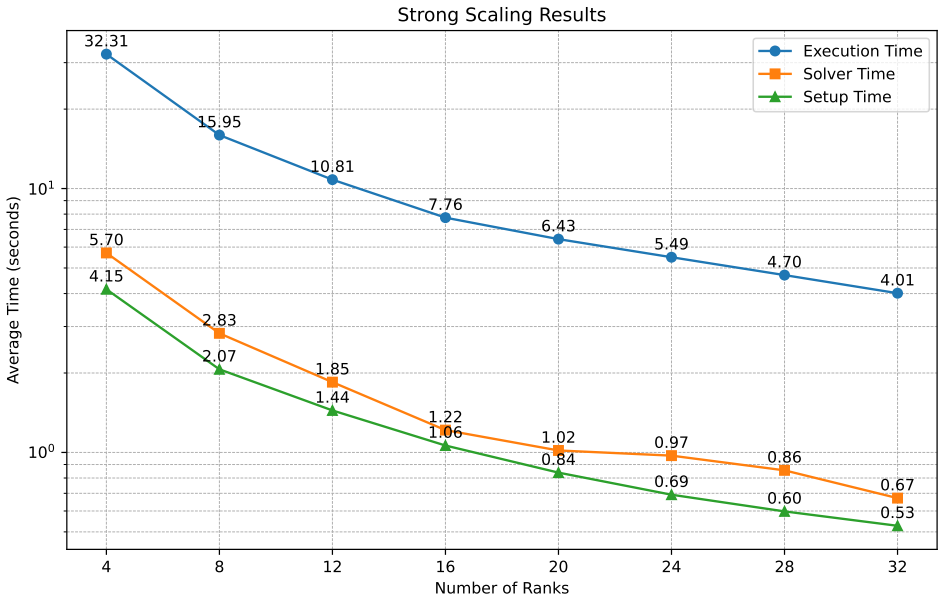

Figure 20: 4096×4096; computed on up to 32 nodes of the EULER VII (Phase 2) cluster with two EPYC_7763 cores (2.4 GHz nominal, 3.7 GHz peak) and 4 Gigabytes of memory per node

The first run tested performance for a 4096×4096 computational grid. Note that the matrix $\underline{\underline{A}}$, while sparse, is actually of dimension $4096^2 \times 4096^2$, requiring more than 8 gigabytes of memory to be processed by PETSc's methods, hence requiring at least 4 cores in this node configuration.

Remarkably, ideal scaling across all methods is measured. This now raises questions towards the single node runs (16, 17, 18, 19), where performance became unstable already lower rank counts than tested in the distributed tests. A potential explanation may be that when running with multiple ranks on the same node, the requested memory might actually be available to all ranks and not just the one it was allocated for. This would also explain why no bad allocations occured at similar problem sizes and rank counts in the single node run.

It should be noted that the maximum amount of memory allocated in the distributed runs is 128GB and not 96 GB as in the single node tests. Thus memory might have been the limiting factor after all.

Now, One should not mistake superior scalability for superior performance. A simple comparison of the absolute execution times between 32 cores distributed 20 and 32 cores single node 16 still puts the single node run ahead by around 15%.

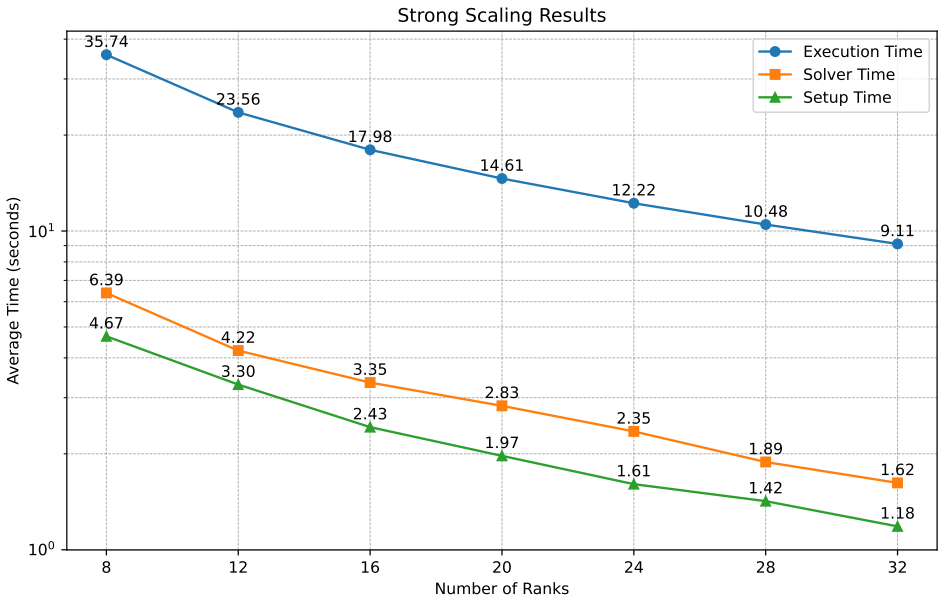

Figure 21: 6144×6144; computed on up to 32 nodes of the EULER VII (Phase 2) cluster with two EPYC_7763 cores (2.4 GHz nominal, 3.7 GHz peak), 4 Gigabytes of memory each

With increasing problem size the memory constraints become even clearer, now requiring at least 8 cores with 32GB of memory on the 6144×6144 computational grid in 21 and 16 cores with 64GB of memory on the 8192×8192 computational grid in 22. This remains to be the case for larger problem sizes such as 10240×10240, but we refrain from considering scaling plots with too little datapoints to observe meaningful scaling results.

Both problem sizes (21, 22) continue to dislay almost ideal scaling (parallel efficiency of over 90% when compared to their respective base case of 8 and 16 ranks) up until the maximum tested rank count of 32. Solely the solver time of the largest distributed test case 22 seems not to improve.

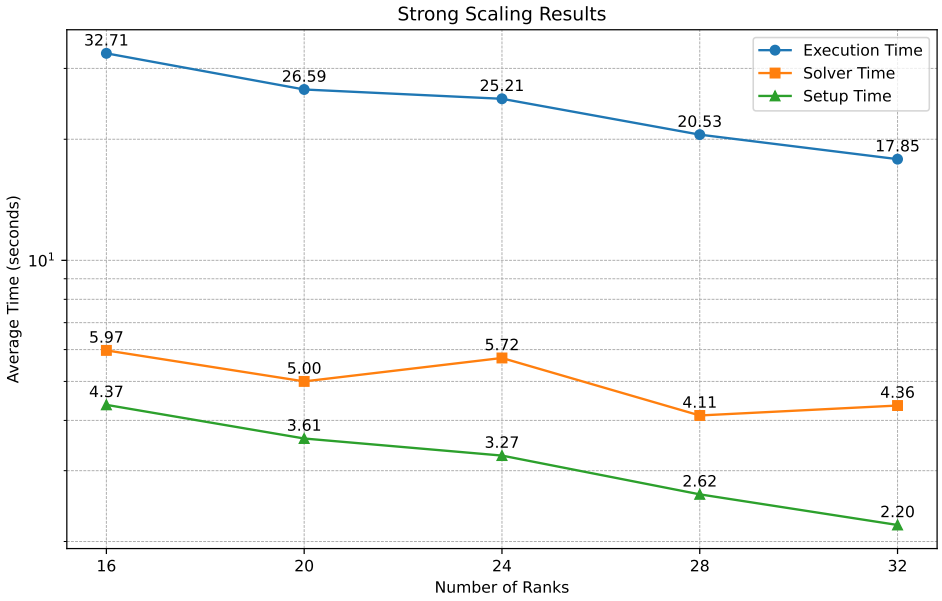

Figure 22: 8192×8192; computed on up to 32 nodes of the EULER VII (Phase 2) cluster with two EPYC_7763 cores (2.4 GHz nominal, 3.7 GHz peak), 4 Gigabytes of memory each