Applied Mathematics & Computational Science 312

High Performance Computing

Problem Set #2 due: March 17, 2025

In this problem set, you solve a linear algebraic system arising from the discretization of Poisson's Equation, a second-order Partial Differential Equation (PDE) based on the Laplacian operator, on a structured grid. It is based upon the Portable, Extensible Toolkit for Scientific computation (PETSc) framework (see `http://www.mcs.anl.gov/petsc`). Once you have described a problem in PETSc's data structures, you may solve it via a wide variety of algorithms, including many of the Colella "dwarves," and on a wide variety of scalable architectures.

Contemporary HPE-Cray systems, including KAUST's Shaheen (`https://www.hpc.kaust.edu.sa/`), are delivered with a vendor-tuned PETSc library against which you may link your application directly. You will build your own instance of the latest PETSc version (3.22.3, as of February 2025) and link the ps2.c example in this exercise against your personal installation.

Although PETSc is the data structure and solver backbone for many scientific and engineering applications seeking scalable performance, it was originally designed by its authors as a toolkit for experimentation on novel solution algorithms. Today, many such algorithms are available through a single, easy-to-use interface, which makes PETSc a useful pedagogical toolkit. It is convenient that a learner may progress to a frontier researcher without outgrowing the package in which they may first experiment with scalable solvers.

PETSc incorporates many of the most useful open source linear and nonlinear numerical solvers in the world through its scalable framework. (This remains so because if something suitable comes along, the PETSc team at Argonne National Laboratory simply extends the interface to include it.) PETSc's installation file comes with configuration scripts that cover most of the commonly available scientific computing systems. It is therefore a convenient one-stop shop for lots of the combinations of software and hardware that you may encounter in a career of using High Performance Computing (HPC) for scientific and engineering simulations.

This problem set is based upon the first three chapters of a 2020 book by Professor Ed Bueler of University of Alaska, Fairbanks, namely *PETSc for Partial Differential Equations*, available in the KAUST digital library. The book is a great resource for learning PETSc. The first three chapters cover this problem set (and much more). It is not necessary to understand them in full to successfully complete this problem set. Nevertheless, this book is an excellent resource, along with the PETSc user manual at `https://petsc.org/release/manual/manual.pdf`, for understanding how to use the framework.

PETSc is written mostly in the C programming language. It is not necessary to have a deep knowledge of either PETSc or C to complete this homework assignment or to do most of the tasks for which you might need a package like PETSc. Nonetheless, the more understanding you acquire, the more facility you will have in obtaining and interpreting results.

Begin this problem set with a review of the PETSc lecture slides (Unit 5) which originate from the 2016 Argonne Training Program in Extreme Scale Computing. (The video of PETSc training at ATPESC'16 is available on-line if you prefer to take the PETSc short course with its developers: `https://www.youtube.com/watch?v=Knk4BqNXCao`.)

You are now a revolution higher on the "spiral" of integrated HPC study. This and the remaining problem sets will specialize on the Newton-Krylov family of solvers on a structured grid, lecture material through Unit 6. We will mix in a variety of Krylov methods and preconditioning methods, and we will add Multigrid as a preconditioner in order to scale. In this problem set, you will play with the number of independent variables in each dimension of one-, two-, and three-dimensional linear scalar problems. In future problem sets, we will generalize from linear to nonlinear. We will also generalize from scalar to several dependent variables (systems of PDEs). We will also be nudged into the setting of multiple physical parameters and do a little modeling.

ps2, provided with this problem set, is a directory containing source code and a makefile, which include the main program and call-back subroutines of a parallel C code that uses PETSc to solve the Poisson Equation. The key line is #45:"`PetscCall(KSPSolve(ksp,b,u))`". Prior to this call, data structures are set up, a subroutine to create the matrix is inserted into the `ksp` solver context, and subroutines for the right-hand side `b` and the exact solution `uexact` (for checking the result) are created. The vector `u` is output from the `KSPSolve` call to compare with `uexact`. In the first computing exercise, you simply invoke this program from the command line.

In the second computing exercise of this problem set, when you will be asked to modify a PETSc example to add features, please follow good software engineering practices by documenting your code with comments that will help another reader (and you later on, after a break!) to understand your code.

You will submit an archived file that contains the following:

- A report containing your responses for every question in the problem set
- Your modified source code file(s)
- `db.petsc` file(s) that contain your PETSc command line options
- Shaheen shell script file(s) to execute test cases that solve the exercises

Please aggregate your submissions in an archived file similar to Listing 1. Here, `YYYYMMDD` is the 8-digit code for the submission date.

```
tar czf 20250317_amcs312_<your_surname>.tar.gz <file1> <file2> ...
```

Listing 1: TAR with GZIP compression instructions

1) Getting started with PETSc [0% (nothing to submit)]

You are welcome to do all of the computing exercises in AMCS 312 on Shaheen-3. However, it may be convenient to run small cases on your own computer. That's what this section is about, but you may skip it if you wish. The exact same source code, with different build options and a greater variety and range of run options will then be portable to a supercomputer. PETSc can be installed on your personal laptop/workstation, and it is compatible with any Unix-like Operating System (e.g. Linux distributions or macOS).

For Microsoft Windows Operating System users, there are two ways to install PETSc: either using a third-party Unix-like environment for Windows (e.g., Cygwin), or installing a Virtual Machine (e.g., Oracle VirtualBox) to run a Unix-like kernel. [Note: Although Windows is supported, we recommend that you use a Unix-like OS, to avoid issues we have not foreseen. The Shaheen supercomputer to which you will graduate is a Linux-based, and this problem set is a step *en route* to running different large-scale demonstration codes on high-performance systems.]

The minimum requirement to run PETSc on your workstation and to carry out this problem set is to have a C compiler (e.g., GNU GCC) installed before installing PETSc. Also, please make sure that the X Window System (X11) is installed and properly working on your system before installing PETSc. X11 is used to provide rudimentary graphical visualizations while PETSc is executing, and upon exit. Other external packages (e.g., MPI) can be downloaded within the PETSc configuration script. However, you can link to an existing installation of these packages if you have them installed already on your system. The details of how download and install PETSc, can be found here: `https://petsc.org/release/install/`

Listing 2 provides the basic download, configuration, installation, and testing commands for PETSc on macOS 15.3.1 using the `Homebrew` software package manager.

```
brew install petsc
export PETSC_DIR=/opt/homebrew/Cellar/petsc/3.22.3
cd $PETSC_DIR/share/petsc/examples/src/snes/tutorials
make ex19
```

Listing 2: Download, configure, install, & test PETSc

If all has gone well with the build, you should see output very similar to Listing 3.

```
mpicc -Wl,-commons,use_dylibs -Wl,-search_paths_first -Wl,-no_compact_unwind
    ↪ -Wl,-no_warn_duplicate_libraries -fPIC -Wall -Wwrite-strings -Wno-unknown-pragmas
    ↪ -Wconversion -Wno-sign-conversion -Wno-float-conversion -Wno-implicit-float-conversion
    ↪ -fstack-protector -fno-stack-check -Qunused-arguments -fvisibility=hidden -Wall
    ↪ -Wwrite-strings -Wno-unknown-pragmas -Wconversion -Wno-sign-conversion
    ↪ -Wno-float-conversion -Wno-implicit-float-conversion -fstack-protector -fno-stack-check
    ↪ -Qunused-arguments -fvisibility=hidden -g -O3
    ↪ -I/opt/homebrew/Cellar/petsc/3.22.3/include -I/opt/homebrew/opt/fftw/include
    ↪ -I/opt/homebrew/opt/metis/include -I/opt/homebrew/opt/hdf5-mpi/include ex19.c
    ↪ -Wl,-rpath,/opt/homebrew/Cellar/petsc/3.22.3/lib -L/opt/homebrew/Cellar/petsc/3.22.3/lib
    ↪ -Wl,-rpath,/opt/homebrew/opt/scalapack/lib -L/opt/homebrew/opt/scalapack/lib
    ↪ -Wl,-rpath,/opt/homebrew/opt/fftw/lib -L/opt/homebrew/opt/fftw/lib
    ↪ -Wl,-rpath,/opt/homebrew/opt/metis/lib -L/opt/homebrew/opt/metis/lib
    ↪ -Wl,-rpath,/opt/homebrew/opt/hdf5-mpi/lib -L/opt/homebrew/opt/hdf5-mpi/lib
    ↪ -Wl,-rpath,/opt/homebrew/Cellar/open-mpi/5.0.6/lib
    ↪ -L/opt/homebrew/Cellar/open-mpi/5.0.6/lib
    ↪ -Wl,-rpath,/opt/homebrew/Cellar/gcc/14.2.0_1/lib/gcc/current/gcc/aarch64-apple-darwin24/14
    ↪ -L/opt/homebrew/Cellar/gcc/14.2.0_1/lib/gcc/current/gcc/aarch64-apple-darwin24/14
    ↪ -Wl,-rpath,/opt/homebrew/Cellar/gcc/14.2.0_1/lib/gcc/current/gcc
    ↪ -L/opt/homebrew/Cellar/gcc/14.2.0_1/lib/gcc/current/gcc
    ↪ -Wl,-rpath,/opt/homebrew/Cellar/gcc/14.2.0_1/lib/gcc/current
    ↪ -L/opt/homebrew/Cellar/gcc/14.2.0_1/lib/gcc/current -lpetsc -lscalapack -lfftw3_mpi
    ↪ -lfftw3 -llapack -lblas -lmetis -lhdf5_hl -lhdf5 -lmpi_usempif08 -lmpi_usempi_ignore_tkr
    ↪ -lmpi_mpifh -lmpi -lgfortran -lemutls_w -lheapt_w -lgfortran -lquadmath -lc++ -o ex19
ld: warning: search path '/opt/homebrew/Cellar/open-mpi/5.0.6/lib' not found
```

Listing 3: ex19 build output

You can now test that the executable itself is running correctly by trying to run it; see Listing 4.

```
./ex19 # You should see output similar to
lid velocity = 0.0625, prandtl # = 1., grashof # = 1.
Number of SNES iterations = 2
```

Listing 4: Run the executable of ex19

2) Getting started with Poisson's Equation on a structured grid [0% (nothing to submit)]

In this problem set, we solve the following Poisson's Equation:

$$-\nabla^2\varphi = f \text{ in } \Omega, \tag{1}$$

$$\varphi = 0 \text{ on } \Gamma, \tag{2}$$

where φ is a potential, f is source term, Ω is the domain, and Γ represents the boundary. The domain Ω can be in 1D (unit line: $0 \le x \le 1$), 2D (unit square: $0 \le x, y \le 1$), and 3D (unit cube: $0 \le x, y, z \le 1$). The boundary Γ is subjected to homogeneous Dirichlet boundary conditions: $\varphi(0) = \varphi(1) = 0$ (in 1D), $\varphi(x,0) = \varphi(x,1) = \varphi(0,y) = \varphi(1,y) = 0$ (in 2D), and $\varphi(x,y,0) = \varphi(x,y,1) = \varphi(x,0,z) = \varphi(x,1,z) = \varphi(0,y,z) = \varphi(1,y,z) = 0$ (in 3D). The Laplacian term in Poisson's Equation (2) can be discretized on a ND structured grid, where $N \in \{1,2,3\}$ by a variety of means, as explored in lecture. Here we discretize it by approximating the second-order partial derivative operators by the second-order centered finite differences. For instance, Equation 3 presents the second-order partial derivative approximation on a 2D grid, where h_x and h_y are the grid spacings in x and y directions, respectively.

$$\nabla^2\varphi_{i,j} = -\frac{\varphi_{i+1,j} - 2\varphi_{i,j} + \varphi_{i-1,j}}{h_x^2} - \frac{\varphi_{i,j+1} - 2\varphi_{i,j} + \varphi_{i,j-1}}{h_y^2}, \tag{3}$$

We can multiply Poisson's Equation (2) by the grid cell area ($h_x \times h_y$) and use Equation (3) at each interior point to get

$$-\frac{h_y}{h_x}(\varphi_{i+1,j} - 2\varphi_{i,j} + \varphi_{i-1,j}) - \frac{h_x}{h_y}(\varphi_{i,j+1} - 2\varphi_{i,j} + \varphi_{i,j-1}) = h_x h_y f_{i,j}. \tag{4}$$

[Note: Scaling by the grid cell area gives matrix coefficients of order unity when the mesh spacings are comparable to each other. If, on the other hand, the x-spacing is much finer, the first term is dominant; if the y-spacing is finer, the second term is dominant.]

In this exercise, we specify the source term, f so that φ satisfies the following exact solutions:

- In 1D:

$$\hat{\varphi}(x) = x^2 - x^4. \tag{5}$$

- In 2D:

$$\hat{\varphi}(x, y) = (x^2 - x^4) \times (y^4 - y^2). \tag{6}$$

- In 3D:

$$\hat{\varphi}(x, y, z) = (x^2 - x^4) \times (y^4 - y^2) \times (z^2 - z^4). \tag{7}$$

This allows us to evaluate the truncation error of the continuous equation in the computational solution and evaluate its convergence as a function of the mesh spacing.

Discretizing Equation (2) in this way generates a linear equation in the form of:

$$Ax = b, \tag{8}$$

where A is a symmetric, positive-definite coefficient matrix, x is the vector of unknowns approximating φ on the grid, and b is the right-hand side vector approximating the values of f. If we consider all values φ to be unknowns, whether on the boundary Γ or in the interior of Ω, we have: 1) m_x unknowns (in 1D) 2) $m_x \times m_y$ unknowns (in 2D), and 3) $m_x \times m_y \times m_z$ unknowns (in 3D).

To build an N-dimensional grid for Equation (2), we use PETSc's DM class for uniform grids. We set the default grid dimensions to 16, but they can be set to different values from the command line. The stencil (star stencil – left, right, center, top, and bottom) is: 3-point stencil (1D), 5-point stencil (2D), and 7-point stencil (3D).

To solve the Poisson's Equation (2), we use the Krylov Subspace (KSP) linear solver object, which is initialized and called from the main program ("call-back" C functions).

We verify the implementation by comparing the numerical solution to the analytical solution (5. 6, and 7). The function computes a norm of the difference between the computed solution and the exact solution (i.e., $||\varphi - \hat{\varphi}||_p$, where $\hat{\varphi}$ is the exact solution). [Note: PETSc supports three types of vector norms: the one-norm $||.||_1$, the two-norm $||.||_2$, and the infinity-norm $||.||_\infty$.]

Now you will draw upon the resources of the PETSc framework as documented online at `https://www.mcs.anl.gov/petsc/`, in the PETSc manual downloadable from this site, and in the book *PETSc for Partial Differential Equations* by Bueler to "dig in" and explore this simple example.

You have been provided with an access to Shaheen at KAUST, on which to try the third exercise. Shaheen is based upon AMD Genoa processors, and the software stack of Shaheen is provided by Cray. In this problem set, you will use Shaheen in all-MPI distributed memory mode, so no explicit threading. (In practice, many users employ hybrid programming paradigms on these systems by using a shared memory programming model, such OpenMP, *inside* of an MPI-based message passing model.) You will want to consult the on-line PETSc manual page to see how to invoke some of the standard algorithmic options you need.

3) Computing Exercise 1 [30%]: *Convergence and performance characteristics of Poisson's Equation*

Please first read, for reference, the `README.md` file to configure, compile, link, and execute the C source code, and then complete the following introductory tasks. Questions about these tasks are welcome in lecture or office hours. They will reinforce various lessons covered in Units 2 (Laplacian and its discretizations), 4 (Structured grids and their implementation), and 5 (PETSc and its solvers).

1. Compile the three versions of the code (1D, 2D, and 3D), and then run them sequentially and document the default KSP and PC options of PETSc of any dimension of your choice, including the relative and absolute tolerances for terminating the iteration. Confirm that the only difference between the default settings of the three versions is the matrix object size, and the size that is reported by PETSc is similar to what Part #2 states. [Note: This includes the matrix dimensions, and the total number of the nonzeros.]

2. For the default grid sizes (16 (1D), 16×16 (2D), $16 \times 16 \times 16$ (3D)), report the wall-clock time, as well as the computational rate (flops/sec) of the most important computational kernels, a norm of the numerical error of your choice, and how many KSP iterations are performed. [Note: Do not be concerned if your performance results thus far are poor. This is a baseline exercise. It is not expected that the code will perform well, since conservative default settings of PETSc's KSP linear solver package are used. In the coming exercises, you will learn variety of ways to improve the performance.]

3. Visualize contour plots that show the structure of A of 1D, 2D, and 3D grid, and write a few sentences to explain the contour plots, and to compare between the three plots. [Note: For accurate visualization, you can dump the structure data into a formatted file, and then use a better visualization tool, e.g., MATLAB, Python, R, ..., to plot the structures.]

4. Run the 1D code, and monitor the KSP residual norm as a function of iteration number and compare the last KSP residual norm with the numerical error norm.

5. Print the evolving estimates of the eigenvalues of the Laplace operator at each KSP iteration of the 3D code.

6. Refine the initial grid of the 2D code four times by a factor of 2 each time. Does the KSP solver converge in the same number of steps? If not, what is the behavior? Does the numerical solution converge to the analytical solution? At what rate? [Hint: The truncation error of the PDE discretization is expected to be reduced by a factor of 4 with each doubling of resolution and halving of h. However, you are expected to exactly observe such behaviors with the max norm only. Hence, to extract the second-order convergence with the other two norms, namely 1-norm and 2-norm, you may need to normalize them, because they represent the algebraic error in the solution, not the truncation error in the discretization. **Therefore, part of the answer of this question is to confirm such observations, do the required modifications to the algebraic error in the solution, and finally explain thoroughly your findings.**]

7. Does the convergence rate of the 2D code implementation match the theory based on condition number? Justify your answer.

8. In the 3D code and for a given grid size, it is possible to demand more performance of the algebraic solver KSP than is worthy of the numerical solution (that is, one can drive down the algebraic residual below the level of the truncation of the discretization, which adds no value). Display this effect in a table by playing with the relative convergence tolerance of the KSP solver, on a few different grid sizes.

9. Equation 3 of Part #2 presents the approximation of the second-order partial derivative operators by the second-order centered finite differences on a 2D grid, whereas Equation 4 shows the same equation after scaling it by the grid cell area ($h_x \times h_y$). Apply the same techniques to approximate the second-order partial derivative operators by the second-order centered finite differences on a 1D and 3D grid. Explain your answer.

4) Computing Exercise 2 [20%]: *Different source terms and different boundary conditions*

(a) Modify the example Poisson's Equation code with source terms derived from the following exact solutions:

1. 1D:
   - $\hat{\varphi}(x) = (x - x^2)$
2. 2D:
   - $\hat{\varphi}(x, y) = (x - x^2) \times (y^2 - y)$
3. 3D:
   - $\hat{\varphi}(x, y, z) = (x - x^2) \times (y^2 - y) \times (z - z^2)$

Compute the numerical error as you refine the grid. Why do you get this behavior?

(b) Modify the example Poisson's Equation code to allow non-homogeneous Dirichlet boundary conditions $\varphi = g$ on Γ, where f and g are derived from the following exact solutions:

1. 1D:
   - $\hat{\varphi}(x) = 3x + \sin(20x)$
2. 2D:
   - $\hat{\varphi}(x, y) = 3x + \sin(20xy)$
3. 3D:
   - $\hat{\varphi}(x, y, z) = 3x + 3z + \sin(20xyz)$

Test if the convergence occurs at the expected rate, and when it occurs. Verify the numerical error as the grid gets refined.

5) Computing Exercise 3 [50%]: *Distributed memory and the SPMD programming paradigm*

This exercise reinforces lessons covered in Unit 1 (Parallel Programming Paradigms).

(a) Refine the initial 2D grid 6 times (use `-da_refine 6`) to create a 1024×1024 Poisson's problem. Using the provided batch script (see details in the Shaheen supplement), execute the code with 64 MPI processes that are compactly allocated across the available compute cores of each allocated node.

[Note: In this problem set, and in this course, we keep the number of requested compute nodes as small as possible by allocating as many cores per node as possible. There are two reasons for this. Batch reservation systems generally favor job requiring fewer compute nodes, ahead of those requiring more, so your job will be scheduled for execution with the smallest delay. In addition, the charging algorithm on most parallel systems charges for the time that nodes are allocated, independent of the number of cores employed per node, so it is usually most cost effective to use all of the cores. If you needed more memory per core, you could use fewer cores per node and allocate more nodes, leaving many of the cores idle on each node, but our exercises are not memory hogs.]

1. Document the different options of the default KSP and PC of PETSc of the parallel code, and compare them with your previously explored sequential one.

2. Is the default Krylov and preconditioner combination the best combination one can recommend in parallel? Recall that the linear system is symmetric and positive definite. Do the default settings of PETSc KSP and PC exploit this structure? The next questions are intended to tune the PETSc KSP and PC parameters.

3. Tabulate the following five performance characteristics for each of the following parametrically defined method: 1) number of linear iterations, 2) error norm, 3) overall runtime, 4) computational rate, and 5) message-passing activities. Document the set of command lines you used. Compare your results with the baseline model you developed initially. Explain your overall performance results and how do you interpret them.

- With the Generalized Minimal Residual (GMRES) method as the Krylov solver, perform the following experiments:
  - Try first GMRES without preconditioning; then try GMRES with block Jacobi as a domain-decomposed preconditioner and Incomplete LU (ILU) on the subdomains.
  - Use the default restart size for the Krylov subspace (30 vectors) at first; then try fewer (15), then greater (60), and finally use 200 vectors. Using the largest subspace should lead to the fewest iterations, but is the execution time proportional to the number of iterations? Reason why or why not.
- Since the matrix is symmetric and positive definite, use the Conjugate Gradient (CG) method as the Krylov solver, and perform the following experiments:
  - Try first CG without preconditioning; then try CG with Jacobi as a domain-decomposed preconditioner.
  - Set a tight relative convergence tolerance for CG and report on the shape of the curve of the residual as a function of iteration count. The matrix two-norm of the error of CG has to be monotonically decreasing (recall the theorem on slide 56 of Unit 4, Part 1). Is the residual norm?
  - Do other Krylov methods for symmetric systems, such as SYMMLQ, offer any advantages?
- Use a direct method for preconditioning, and set Krylov to apply the preconditioner only. Perform the following experiments.
  - Try LU factorization direct solver, as a preconditioner.
  - Try Cholesky factorization direct solver, as a preconditioner; then vary the matrix ordering techniques: 1) Nested Dissection, 2) Reverse Cuthill-McKee.

4. Compare the performance of unpreconditioned and Jacobi-preconditioned CG, by refining the initial grid 0, 2, 4, and 6 times. Explain how that ICC[0] preconditioning gives lower iteration counts but the same scaling in h. [Note: You may need to adjust the number of MPI processes based upon the grid size, so that you do not get an MPI error because of the grid is too coarse to distribute across the cores. For example, in the case of 0 do not run 64 MPI processes, where you have only 16 points in each grid direction.]

5. Lectures on multigrid as a solver and preconditioner will follow, but you can play with the method now without effort. Once you have been amazed by multigrid, you will be motivated to understand its analysis. Use the same settings as the previous question with CG iterative method as a Krylov solve. Perform the following experiments and compare your results with your previous results of GMRES and CG.

   - Use PETSc implementation of the Geometric Algebraic Multigrid as a preconditioner inside CG with the default settings.
   - Reduce the default number of levels of multigrid to 2, 4, and 12.
   - Set the number of multigrid levels to 3, and at each of the different levels change the inner Krylov solver and the preconditioner to one of your choice. [Note: This means that you should have 3 different Krylov solvers and 3 different preconditioners, one for each level.]
   - Set the cycle type of multigrid to the W cycle.
   - Change the default multigrid type to additive and then full.
   - Try Galerkin process to compute the coarse operators.
   - Try the Hypre implementation of the Algebraic Multigrid (BoomerAMG) with the default settings. [Note: Hypre is an external package implemented by Lawrence Livermore National Laboratory (LLNL) and it is not part of PETSc package. However, since this exercise is carried out on Shaheen system, the PETSc installation of Cray on the system includes Hypre and it automatically links to the library when you compile a PETSc code. You can always configure your own PETSc installation with Hypre package by simply adding `--download-hypre` option to the PETSc configuration script.]

(b) Refine the initial $16 \times 16 \times 16$ 3D grid 3 times to create a $128 \times 128 \times 128$ Poisson problem. Execute the code with the same processing resources as before (64 MPI ranks), with CG as Krylov solver, Block-Jacobi as a domain decomposed preconditioner, and ICC as a sub-domain preconditioner. Test the convergence of the code. How many linear iterations does KSP preform? Now, rerun the same problem with CG as Krylov solve, and Algebraic Multigrid (BoomerAMG) as preconditioner. Compare the KSP convergence results of "CG plus ICC" with "CG plus multigrid". Interpret your results. How many linear iterations does KSP perform compared to the previous case? Compare the overall average runtime. Refine the grid again 4, 5, and 6 (a billion unknowns) times in each direction with Algebraic Multigrid (BoomerAMG) as a preconditioner and CG as a Krylov solver. Test the scaling of multigrid in terms of number of the linear iterations as the grid gets refined. What do you observe?