Introduction to High Performance Computing

Problem Set #2

In this problem set, you solve a linear algebraic system arising from the discretization of Poisson's Equation, a second-order Partial Differential Equations (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 Cray systems, including Blue Waters and KAUST's Shaheen-2, are delivered with a vendor-tuned PETSc library against which you may link your application directly. Nevertheless, you may wish to build your own version or link against an installation built just for this course, for course-customized features, or because there are new features in the latest PETSc release that are available ahead of the vendor's latest supported version. PETSc is always under development because the ecosystem in which it thrives is a dynamic one and invites or requires adaptation.

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 is 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 a new book by Professor Ed Bueler of University of Alaska, Fairbanks, namely *PETSc for Partial Differential Equations*, available as of the time of posting this assignment at `https://github.com/bueler/p4pdes/releases/download/v13aug2016-draft/book.pdf`, and soon to be published by SIAM. The book is a great resource for learning PETSc. It is not complete yet; however, well edited drafts of the first seven chapters have been released. The first three chapters cover this problem set (and much more). You are not required to understand them in full to successfully solve this problem set. Nevertheless, this book is the an excellent resource, along with the PETSc user manual at `http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf`, for understanding how to use the framework.

PETSc is mostly written in the C programming language. You are not required 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.

In parts 2 and 3 of this assignment, when you 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 source code
- A `Makefile` that includes targets to compile and build your source code
- A `README.md` file that explains how to compile, build, and run your source code
- Shell script files 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 YYYYMMDD_Intro2HPC_<your_surname>.tar.gz <file1> <file2> ...
```

Listing 1: Tar with Gzip compression instructions

1) Getting Started with PETSc [nothing to submit from this part]

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 forseen. The Blue Waters and Shaheen-2 computers to which you will graduate are Linux-based, and this problem set is a step *en route* to running different large-scale demonstration codes on these 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. Listing 2 provides the basic download, configuration, and installation commands for PETSc using the Unix Command-Line Interface (CLI).

```
1 cd $HOME
2 wget http://ftp.mcs.anl.gov/pub/petsc/release-snapshots/petsc-3.7.3.tar.gz
3 tar xvzf petsc-3.7.3.tar.gz
4 cd $HOME/petsc-3.7.3/
5 ./configure --download-f2cblaslapack --download-mpich
6 make all test
```

Listing 2: Download, configure, and install PETSc

Here, "`--download-f2cblaslapack`" and "`--download-mpich`" are flags that instruct the configuration script to download BLAS and LAPACK packages for linear algebra routines, and MPICH implementation of MPI for distributed-memory routines, respectively; these are standard building blocks for PETSc and numerous other HPC frameworks. [Note: If you have a Fortran compiler installed on your system, then you may use "`--download-fblaslapack`" flag instead of "`--download-f2cblaslapack`". Also, if `wget`[1] is not installed on your system or it is not compatible by default with your system (e.g., macOS), then you can either use any other software with the same functionality (e.g., `curl` for macOS) or, of course, you may simply copy-and-paste the HTTP link in line 2 of Listing 2 into your Internet Browser to download the compressed tar file and move it into your home directory.] Note that version 3.7.3 of PETSc, released on 24 July 2016, is approximately 22 MB in size.

The configuration script assumes that the compilers are properly set in your default Linux path. Otherwise, you have to provide the compilers' paths using flags to point to for each specific compiler (e.g., `--with-cc=<path/to/C/compiler>`). All of this information and more can be found in the installation web-page of PETSc: `http://www.mcs.anl.gov/petsc/documentation/installation.html`. Please refer to it for further assistance.

After the PETSc installation finishes, you need to set the PETSc environmental variables in your `.bashrc` file. See Listing 3 to do this using the `vi` editor, or use another editor of your choice. [Note: For macOS users, you may add them in your `.bash_profile` instead of `.bashrc`].

```
vi + $HOME/.bashrc
# Hit o key to insert a new line after the current line (switch to the edit mode)
```

[1]`wget` is a software that retrieves contents from web servers. It supports HTTP, HTTPS, and FTP protocols.

```
# Insert the following two lines
export PETSC_DIR=$HOME/petsc-3.7.3/
export PETSC_ARCH=arch-linux2-c-debug
# Hit Esc key to exit the edit mode
# Type :wq and hit Enter (the Return Key) to write, save, and quit
source $HOME/.bashrc
```

Listing 3: Set PETSc's environment variables

Ensure that the PETSC_DIR and PETSC_ARCH environmental variables are properly set by echoing them. See Listing 4.

```
echo $PETSC_DIR; echo $PETSC_ARCH
```

Listing 4: Echo PETSc's environment variables

If they are properly set, then you should see some outputs similar to Listing 5.

```
/home/<your_user_name>/petsc-3.7.3
arch-linux2-c-debug
```

Listing 5: Values of PETSC_DIR and PETSC_ARCH

Follow the instructions of Listing 6 to build a PETSc example to make sure that your PETSc installation is working well.

```
cd $PETSC_DIR/src/snes/examples/tutorials/
make ex19
```

Listing 6: Build PETSc SNES e19

If all has gone well with the build, you should see output very similar to Listing 7.

```
/home/<user_name>/petsc-3.7.3/arch-linux2-c-debug/bin/mpicc -o ex19.o
-c -Wall -Wwrite-strings -Wno-strict-aliasing -Wno-unknown-pragmas
-fvisibility=hidden -g3 -I/home/<user_name>/petsc-3.7.3/include
-I/home/<user_name>/petsc-3.7.3/arch-linux2-c-debug/include
‘pwd‘/ex19.c
/home/<user_name>/petsc-3.7.3/arch-linux2-c-debug/bin/mpicc
-Wall -Wwrite-strings -Wno-strict-aliasing -Wno-unknown-pragmas
-fvisibility=hidden -g3 -o ex19
ex19.o -Wl,-rpath,/home/<user_name>/petsc-3.7.3/arch-linux2-c-debug/lib
-L/home/<user_name>/petsc-3.7.3/arch-linux2-c-debug/lib
-lpetsc -Wl,-rpath,/home/<user_name>/petsc-3.7.3/arch-linux2-c-debug/lib
-lflapack -lfblas -lpthread -lX11 -lm -Wl,-rpath,/usr/lib/gcc/x86_64-linux-gnu/5
-L/usr/lib/gcc/x86_64-linux-gnu/5 -Wl,-rpath,/usr/lib/x86_64-linux-gnu
-L/usr/lib/x86_64-linux-gnu -Wl,-rpath,/lib/x86_64-linux-gnu -L/lib/x86_64-linux-gnu
-lmpifort -lgfortran -lm -lgfortran -lm -lquadmath -lm -lmpicxx -lstdc++
-Wl,-rpath,/home/<user_name>/petsc-3.7.3/arch-linux2-c-debug/lib
-L/home/<user_name>/petsc-3.7.3/arch-linux2-c-debug/lib
-Wl,-rpath,/usr/lib/gcc/x86_64-linux-gnu/5 -L/usr/lib/gcc/x86_64-linux-gnu/5
-Wl,-rpath,/usr/lib/x86_64-linux-gnu -L/usr/lib/x86_64-linux-gnu
-Wl,-rpath,/lib/x86_64-linux-gnu -L/lib/x86_64-linux-gnu
-Wl,-rpath,/usr/lib/x86_64-linux-gnu -L/usr/lib/x86_64-linux-gnu
-ldl -Wl,-rpath,/home/<user_name>/petsc-3.7.3/arch-linux2-c-debug/lib
-lmpi -lgcc_s -ldl
/bin/rm -f ex19.o
```

Listing 7: ex19 build output

You can now test that the executable itself is running correctly by trying to run it; see Listing 8.

```
# For sequential run
./ex19
# For parallel run
/home/<user_name>/petsc-3.7.3/arch-linux2-c-debug/bin/mpirun \
-np <number_of_mpi_ranks> ./ex19
```

Listing 8: Run the executable of ex19

You should see output similar to Listing 9.

```
lid velocity = 0.0625, prandtl = 1., grashof = 1.
Number of SNES iterations = 2
```

Listing 9: ex19 run output

2) Getting started with Poisson's Equation on a 2D structured grid

`ps2.c`, provided with this assignment, is the main program of a parallel C code that uses PETSc[2] to solve Poisson's equation:

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

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

where $\varphi(x, y)$ is a potential and $f(x, y)$ is source term. The domain Ω is the unit square: $0 \leq x, y \leq 1$, and the boundary Γ is subject to homogeneous Dirichlet boundary conditions: $\varphi(x, 0) = \varphi(x, 1) = \varphi(0, y) = \varphi(1, y) = 0$.

The Laplacian term in Poisson's Equation (2) can be discretized on a 2D structured grid by a variety of means, as explored in lecture. See Figure 1 for the definitions of various index conventions. Approximating the second-order partial derivative operators by the second-order centered finite differences yields:

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

where h_x and h_y are the grid spacings in x and y directions, respectively. (These grid spacings are set at runtime, either by default or command-line specification, based on the number of subintervals into which the x and y axes are discretized.)

We can multiply equation 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. If, on the other hand, the x-spacing is much smaller, the first term is dominant; if the y-spacing is much smaller, the second term is dominant.]

In this exercise, we specify the source term, $f(x, y)$ so that $\varphi(x, y)$ satisfies the following exact solution:

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

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.

[2] To build and execute ps2.c, please follow the instructions in the `README.md` file.

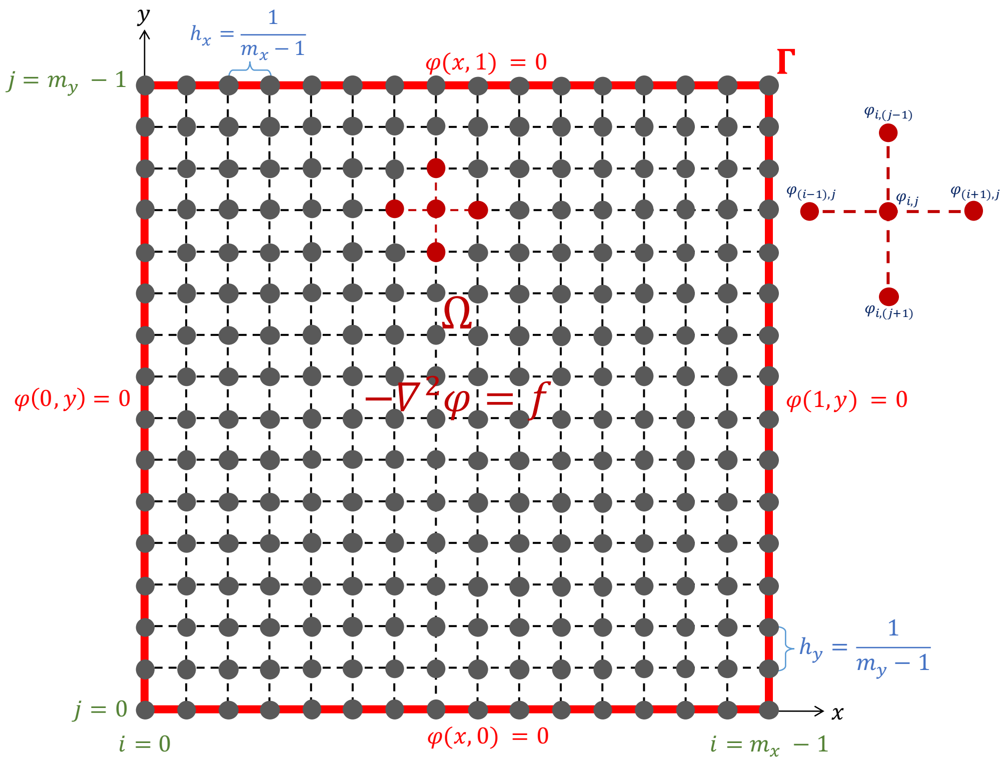

Figure 1: 2D structured grid (16×16); the grid coordinates are $(x_i, y_j) = (i \times h_x, j \times h_y)$

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

$$Ax = b, \tag{6}$$

where A is a symmetric, positive-definite coefficient matrix, u is the vector of unknowns approximating $\varphi_{i,j}$ on the grid, and b is the right-hand side vector approximating the values of $f_{i,j}$. If we consider all values $\varphi_{i,j}$ to be unknowns, whether on the boundary Γ or in the interior of Ω, we have $m_x \times m_y$ unknowns.

To build a two-dimensional grid for Equation (2), we use PETSc's DM class for uniform grids. As shown in Figure 1, we set the default grid dimensions to 16×16, but they can be set to different values from the command line. As in Figure 1, the stencil is a 2D 5-point stencil that relates 5 unknowns: center ($\varphi_{i,j}$), south ($\varphi_{i,j-1}$), north ($\varphi_{i,j+1}$), west ($\varphi_{i-1,j}$), and east ($\varphi_{i+1,j}$). For distributed-memory implementation, each MPI rank stores an extra layer of ghost nodes to communicate with its neighbors.

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. To use KSP, we need two user-defined "call-back" C functions, as follows:

- `compute_rhs(KSP, Vec, void *)`: this function computes the right-hand side vector b. It takes a PETSc KSP object, a right-hand side PETSc vector, and a user-defined C struct (optional)
- `compute_opt(KSP, Mat, Mat, void *)`: this function computes and assembles the coefficient matrix A. It takes a PETSc KSP object, an operator PETSc matrix, a preconditioning PETSc matrix, and a user-defined C struct (optional)

These functions are passed as arguments to predefined PETSc functions `KSPSetComputeRHS()` and `KSPSetComputeOperators()`, respectively.

The C function (`test_convergence_rate(KSP, Vec)`) verifies the implementation by comparing the numerical solution to the analytical solution (5). 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 the convergence and performance characteristics of this simple example.

1. Run the code sequentially and document the default KSP and PC options of PETSc, including the relative and absolute tolerances for terminating the iteration.
2. For the default 16×16 grid, report the wall-clock time, as well as the computational rate (flops/sec) of the most important computational kernels, the two-norm of the numerical error, and how many KSP iterations are performed.
3. Visualize a contour plot that shows the structure of A and write a few sentences to explain the contour plot.
4. 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.
6. Refine the initial grid three 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?
7. Does the convergence rate of your implementation match the theory based on condition number? Justify your answer.
8. For a given grid size, it is possible to demand more performance of the algebraic solver KSP than is worthy of the numerical solution. Display this effect in a table by playing with the relative convergence tolerance of the KSP solver, on a few different grid sizes.
9. Run the code "in parallel" with 2 and 4 MPI processes. (Depending upon the number of cores on your system, you may or may not realize actual parallel performance, but PETSc can emulate multiple-processor performance as an aid to debugging.) You can go beyond 4 processes if you have enough computational resources. With a large enough grid size, compare the runtimes of the executions with different numbers of processors (1, 2, and 4) and note the speedup with respect to 1 processor. Also, compute the parallel efficiency, which is the speedup divided by the number of processors.

[Note: Do not be concerned if your performance results thus far are poor. This is a baseline exercise. It is not expected that your code will perform well, since conservative default settings of PETSc's KSP linear solver package are used. In the coming problem sets, you will learn variety of ways to improve the performance.]

3) Poisson's Equation with different source terms (from Bueler, Ch. 3)

Modify the example Poisson code with source terms derived from the following exact solution:

$$\hat{\varphi}(x, y) = (x - x^2)(y^2 - y)$$

Compute the numerical error as you refine the grid. What behavior do you see, and why do you get this behavior?

4) Poisson's Equation with Different Boundary Conditions (from Bueler, Ch. 3)

Modify your implementation to allow non-homogeneous Dirichlet boundary conditions $\varphi = g$ on Γ, where f and g are derived from the following exact solution $\hat{\varphi}(x, y) = 3x + \sin(20xy)$. Test if the convergence occurs at the expected rate, and when it occurs. Verify the numerical error as the grid gets refined.