Introduction to High Performance Computing

Problem Set #4

This problem set is built upon Problem Set #2, in which you solved a linear algebraic system arising from the discretization of Poisson's Equation, a second-order PDE based on the Laplacian operator, on a structured grid in two dimensions.

Begin this problem set with a review of the previous one and 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 now available on-line if you prefer to take the PETSc short course with its developers.)

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 structured grids, Unit 6 or earlier. 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 (considering also one-dimensional and three-dimensional problems). In future problem sets, we will generalize from linear to nonlinear. We will also generalize from one to several dependent variables (systems of PDEs). And we will be nudged into the setting of multiple physical parameters and do a little modeling.

In Problem Set #2, you solved a 2D PDE discretized as finely as 256×256 on a local workstation. In this problem set, you solve the same Poisson problem on finer resolutions in parallel and on many cores, while exploring a broad range of linear solvers and precondtioners provided by PETSc.

`ps4.c`, provided with this assignment, is a file containing the main program and call-back subroutines of a parallel C code that uses PETSc to solve the Poisson equation of Problem Set #2. The code is almost the same as the `ps2.c` code, with minor improvements. The difference in this assignment is the computer.

You have been provided with an access to either Blue Waters at UIUC or Shaheen at KAUST, on which to try these exercises. Both systems are of Cray architecture, and they run almost the same software stacks provided by Cray. Blue Waters is based on AMD processors and Shaheen on Intel, but their internal differences are not significant, since we use both systems in all-MPI distributed memory mode in these PETSc exercises. (In practice, many users employ hybrid programming models on these systems by using a shared memory programming model, such as OpenMP, *inside* of an MPI-based message passing model.) Unlike the exercises on your workstation, you do not need to install PETSc. We recommend using the Cray version of PETSc that is already installed on both systems. It is usually a version behind what you can pull from Argonne; however, you do not need any extra features that are available on in newer releases for these exercises. You will want to consult the online PETSc manual pages to see how to invoke some of the standard algorithmic options you need. A separate one-page document that is available for distribution along with this problem set describes how to use one system or the other, with a sample job script to help you getting started running jobs in distributed memory.

1) 2D Poisson's Equation on a distributed memory system in the SPMD programming model

Refine the initial 17×17 grid created by default in `ps4.c` 6 times (use `-da_refine 6`) to create a 1025×1025 Poisson problem (a million unknowns is not considered a large problem in 2016). Using the provided batch script (see details in the Blue Waters or Shaheen supplement), execute the code with 64 MPI processes that are compactly allocated across the available compute cores of each allocated node. For instance, for a dual-socket node with 16 cores per socket, assign 32 MPI ranks per node (16 ranks for each socket), and allocate only 2 compute nodes of the Cray system.

[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.]

(a) Baseline: default PETSc KSP and PC settings

1. What are the default KSP options of PETSc? Recall the `-ksp_view` execution-time switch you used in Problem Set #2. What are the key differences between the default PETSc options of KSP in sequential and in parallel? Explain these options and their functionalities in the context of parallel processing.
2. With the default KSP options, does the KSP converge or diverge? How many KSP iterations are performed?
3. What is the maximum norm of the overall error ($||\varphi - \hat{\varphi}||_\infty$)?
4. Turn on the appropriate monitoring and report the overall runtime, the effective computational rate, and the message-passing activity.

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

(b) PETSc linear solvers and preconditioners

1. With the Generalized Minimal Residual (GMRES) method as the Krylov solver, and 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.
   - With additive Schwarz domain-decomposed preconditioning with ILU[0] as a subdomain preconditioner; try different Schwarz overlap levels: 0 (no overlap) and 3. Again, which is better in iteration count, and is the reduction in execution time proportional? Reason why or why not?
   - Use the default ILU level of fill (ILU[0]) at first; then try two higher levels of fill: level 1 and 3. Higher fill should lead to fewer iterations, but is the execution time proportional to the number of iterations? Is there a point of diminishing returns on execution time? Reason why or why not.
   - Try varying subdomain overlap and ILU fill level to find the best pair in terms of execution time.
   - Try one KSP accelerator besides GMRES, such as BiCG-Stabilized? Do your observations and parameter choices hold?

2. 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 and Incomplete Cholesky (ICC) on the subdomains.
   - 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?
   - With additive Schwarz domain-decomposed preconditioning with ICC[0] as a subdomain preconditioner; try different Schwarz overlap levels: 0 (no overlap) and 3. Again, which is better in iteration count, and is the reduction in execution time proportional? Do any of your recommendations based on GMRES experience change?

- Use the default ICC level of fill (ICC[0]) at first; then try higher fill of level 1 and 3. Again, is there a point of diminishing returns?
- Do other Krylov methods for symmetric systems, such as SYMMLQ, offer any advantages?

3. 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; then vary the matrix ordering techniques: 1) Nested Dissection, 2) Reverse Cuthill-Mckee.
   - Try Cholesky factorization direct solver, as a preconditioner; then vary the matrix ordering techniques: 1) Nested Dissection, 2) Reverse Cuthill-Mckee.

Tabulate the following five performance characteristics for each parametrically defined method: 1) number of linear iterations, 2) error norm, 3) overall runtime, 4) computational rate, and 5) message-passing activity. 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.

2) Preconditioned vs. unpreconditioned CG

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.]

3) Multigrid as a preconditioner

Lectures on multigrid as a solver and preconditioner will follow, but you can play with the methods now without effort. Once you have been amazed by multigrid, you will be motivated to understand its analysis.

(a) Algebraic and Geometric Multigrid

Use the same settings as the previous question with Conjugate Gradient (CG) iterative method as a Krylov solve. Perform the following experiments.

1. Use PETSc implementation of the Algebraic Multigrid as a preconditioner inside CG with the default settings.
2. Reduce the default number of levels of multigrid to 2, 4, and 12.
3. 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.]
4. Set the cycle type of multigrid to the **W** cycle.
5. Change the default multigrid type to additive and then full.
6. Try Galerkin process to compute the coarse operators.
7. Use PETSc implementation of the Geometric Multigrid as a preconditioner inside CG with the default settings.
8. 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 problem set is carried out on either Shaheen or Blue Waters systems, the PETSc installation of Cray on these two systems includes Hypre and it automatically links to the library when you compiler 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.]

Compare your results with the your previous results of GMRES and CG.

(b) Scaling of Multigrid

Refine the initial 17×17 grid 7 times to create a 2049×2049 Poisson problem. Execute the code with the same processing resources as before (64 MPI ranks), with CG as Krylov solver, 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 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 6, 8, and 10 times in each direction with Algebraic Multigrid 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?

4) Coding exercise

You do not need to use a supercomputer to carry out the following two exercises. You can use your workstation. (If you want to use a supercomputer, implement on your workstation first, debug there, and then run on a supercomputer.)

(a) One Dimension

Modify the `ps4.c` code to solve the Poisson's Equation in 1D, with homogeneous Dirichlet boundary conditions in 1D. The source term is derived from the following exact solution: $\varpi(x) = x^2 - x^4$.

Run in a sequence of grid refinements to show that your implementation converges.

(b) Three Dimensions

Modify the `ps4.c` code to solve the Poisson's Equation in 3D domain, with homogeneous Dirichlet boundary conditions in 3D. The source term is derived from the following exact solution: $\varpi(x, y, z) = (x^2 - x^4) \times (y^4 - y^2) \times (z^2 - z^4)$.

Run in a sequence of grid refinements to show that your implementation converges.