MPCS 51087
Problem Set 4

Spring 2018

1 Distributed Conjugate Gradient Method

Due Date: Monday, May 14, 2018 by 11:59 pm

1.1 Poisson's Equation in Matrix Form

In this problem set you will implement a parallel conjugate gradient solver and apply it to the solution of the 2D Poisson equation. The Poisson equation is a steady state version of our familiar heat equation. Poisson applies to a huge range of phenomena including steady state head conduction, gravity, electrostatics, potential flow, etc.

Poisson's equation can be written as our heat equation but without the time derivative:

$$\nabla^2 T = b \tag{1}$$

or in 2D Cartesian space as:

$$\frac{\partial^2 T(x,y)}{\partial x^2} + \frac{\partial^2 T(x,y)}{\partial y^2} = b(x,y) \tag{2}$$

where T is some value of interest we wish to solve for (e.g., temperature) on a 2D domain and b is a specified source term.

Similar to the approach we took in problem set 1, we use a finite difference approximation to convert the continuous PDE into a discrete computational grid:

$$\frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{\Delta x^2} + \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j+1}}{\Delta y^2} = b_{i,j} \tag{3}$$

If we assume that $\Delta x^2 = \Delta y^2 \equiv \Delta^2$, and we redefine $b(x,y)$ as $b(x,y)\Delta^2$ (i.e. absorb Δ into the definition of b), we can rewrite Equation 3 as:

$$-T_{i,j-1} - T_{i-1,j} + 4T_{i,j} - T_{i+1,j} - T_{i,j+1} = b_{i,j} \tag{4}$$

This forms a set of linear equations that can be written in matrix-vector form as:

$$Ax = b \tag{5}$$

where for an $n \times n$ physical domain A is the operator matrix of size $n^2 \times n^2$, x is the solution vector of length n^2, and b is the source term vector of length n^2. That is, 2D arrays are flattened into 1D to represent them as vectors.

For example, a problem with $n = 4$ and a source term b of zero would yield the following values for A and b:

$$A = \begin{bmatrix}
4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4
\end{bmatrix}$$

The x vector stores the T array as follows:

$$x = \begin{bmatrix} T_{0,0} & T_{0,1} & T_{0,2} & T_{0,3} & T_{1,0} & T_{1,1} & \dots & T_{3,3} \end{bmatrix}^T$$

and a b vector indicating a zero source term can be written as follows:

$$b = \begin{bmatrix} 0 & \dots & 0 \end{bmatrix}^T$$

We have now defined our linear system from Equation 5 completely, so all that's left is to solve for x.

1.2 Right Hand Side Source Term

In the example b vector given in the previous section, b was set as being completely zero which indicates that there is no source term present in the problem. However, the right hand side (RHS) b vector can also contain a fixed source term that can be applied to anywhere within the problem domain, or could also be used to apply a boundary condition around the edges of the problem. In this assignment, we will consider a small circular non-zero source term in the center of the domain and a value of zero elsewhere, as shown in Figure 1 (a). When we use this source term for Poisson's equation, we should converge to the solution shown in Figure 1 (b).

1.3 Conjugate Gradient Algorithm

Completing this assignment does not require understanding the theoretical underpinnings of CG, only knowing that it is an iterative method that solves $Ax = b$ for any symmetric, positive definite choice of A (both properties of which hold in this case). Pseudocode for the CG method is listed in Algorithm 1 below.

Algorithm 1 can be implemented in serial in a very literal manner in matrix based languages such as Matlab and Julia. However, HPC languages on the other hand require you to define and implement several basic linear algebra functions:

- matvec - A matrix-vector product function that finds the solution of some 2D matrix M and a vector w, and stores the solution in a vector v, as in $v = M * w$
- dotp - A dot product function that finds the dot product of two vectors and returns a scalar value, as in $a = b \cdot c$, where a is a scalar and b and c are vectors
- axpy - A vector addition function that scales and adds two vectors, i.e., $w = \alpha w + \beta v$ where w and v are vectors and α and β are scalars.

With the above functions defined, you can easily implement a general conjugate gradient solver in serial for any symmetric positive definite choice of input A. As we discuss in subsection 2.1, we will be giving you a reference implementation of conjugate gradient in C, so you won't have to implement these functions yourself if you don't want to.

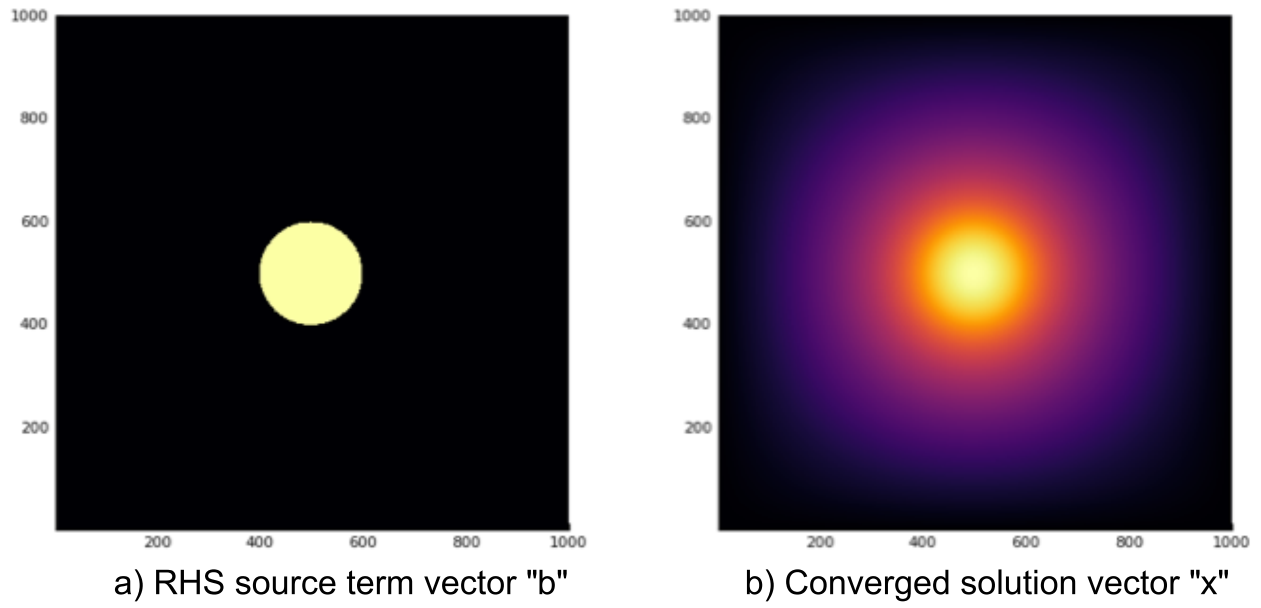

Figure 1: An example of source term vector b and converged solution vector x.

1.4 Sparse Conjugate Gradient Algorithm

The key for efficient implementation of Algorithm 1 is efficient representation of the A matrix. If we implement the method in a naive manner, storing every element of A explicitly, we would be subject to the impractically huge memory requirements of storing a dense $n^2 \times n^2$ A matrix. Therefore, in practice we need to optimize our solver to account for the fact that A is actually a very sparse matrix and therefore does not need to be stored in a dense format. While we could simply use a sparse matrix format library to convert the dense matrix into a sparse form that only stores non-zero indices (or could code up a generic sparse representation ourselves), we can actually do even better than this by capturing the action of the matrix in an "on the fly" manner without storing it at all. If we inspect how we actually use the A matrix in Algorithm 1, we find that it only ever gets accessed on lines 6 and 13, where it is only used as part of a matrix-vector product (matvec).

Therefore, to capture the action of the A 2D Poisson matrix operator without storing it in memory at all, we only need to be able to solve $v = Aw$. That is, we just need to come up with a way determining the correct v vector from a known w vector. We can leverage the fact that A for the 2D Poisson operator is not a random matrix, but has a very simple pattern that we can implement with a few rules in code to capture the same effect of the matrix. To figure out what the rules we need to code are, we can look at the left hand side of Equation 4, which defines the relationship between w and v in the matvec operation. When converted from 2D to 1D notation, the equation becomes:

$$v[i] = -w[i-n] - w[i-1] + 4w[i] - w[i+1] - w[i+n] \tag{6}$$

So, for a known w, we can simply use Equation 6 to compute v for all indices, with a few exceptions. Notably, this new on-the-fly matvec routine still needs to account for the 2D block patterns that we've encoded in A, so there will be some logic required when implementing to ensure that the proper terms are used. That is, Equation 6 assumes that all rows have the same 5 terms, when in reality some rows will have fewer than 5 terms due to where they are located in the 2D physical domain and will need to be encoded as such. For instance, an index that correponds to being located on the far right edge of the domain should not be reading from the $w[i+1]$ term, as it has no right side neighbor.

With this optimized "on-the-fly" (sparse) matvec routine, we can greatly accelerate our CG solver by simply replacing the old matvec calls with the new on-the-fly matvec function. As we will discuss in subsection 2.1, we will be giving you a reference implementation of conjugate gradient in C, so you won't have to

Algorithm 1 Conjugate Gradient Pseudocode

```
1: Input $n$, $A$, $b$                                  ▷ n = Physical domain dimension
2: $N = n^2$
3: Allocate $x$                                         ▷ Length $N$ 1D Solution vector
4: $x = 0$
5: Allocate $r$                                         ▷ Length $N$ 1D vector used in CG
6: $r = b - Ax$
7: Allocate $p$                                         ▷ Length $N$ 1D vector used in CG
8: $p = r$
9: Allocate $z$                                         ▷ Length $N$ 1D vector used in CG
10: rsold $= r^T \cdot r$
11: Max Iterations $= N$
12: for $1 \leq i \leq$ Max Iterations do
13:     $z = Ap$
14:     $\alpha =$rsold$/(p^T \cdot z)$
15:     $x = x + \alpha p$
16:     $r = r - \alpha z$
17:     rsnew $= r^T \cdot r$
18:     if $\sqrt{\text{rsnew}} \leq$ 1e-10 then             ▷ Test for Convergence
19:         break
20:     end if
21:     $p = r +$ (rsnew/rsold)$p$
22:     rsold = rsnew
23: end for
24: Return $x$
```

implement these functions yourself if you don't want to, but we are discussing them in depth here as basic knowledge of how the method works and how data is accessed is critical for understanding how to parallelize the method.

2 Serial

2.1 Serial Implementation (10 points)

Your task for this section is to write a serial code in an HPC language that implements a sparse "on-the-fly" CG solver for the 2D Poisson Equation. To make this section extremely easy, we have provided you with a completed C version of the serial CG solver which is available at `https://github.com/jtramm/distributed_cg`. You are welcome to use this code and turn it in verbatim for this section, convert it to another HPC language of your choice (C++, Fortran), or write your own version from scratch. In subsequent sections of this assignment, you will be tasked with parallelizing the solver with MPI. You do not need to use the provided C code. If you do use it, you are free to change any of the functional interfaces, program structure, etc to meet your needs as you see fit.

If you decide to implement your own version from scratch, you'll need to have enough functionality to run a variably sized sparse "on-the-fly" CG solve for the 2D Poisson equation. You'll also need to have the capability to create a RHS b vector in a similar pattern to what we have in the example C code, as defined in the function `find_b()`.

For reference and comparison purposes, the example C code also implements the inefficient "dense" solver method wherein the 'A' matrix is fully stored in dense format in memory. This is not a practical method, but was included so it's clear how A is formed and how the naive version of the linear solve works if you are not familiar with linear algebra.

2.2 Serial Experimentation (10 points)

In this section, you will be using the example C code to do some basic performance analysis to make sure it is clear what the benefit of using the sparse solve method is. If you have implemented your own sparse CG solver, you can still use the example C code to test against to make sure your code's performance looks reasonable.

Compile and run the example C solver in serial, testing both the sparse and dense solvers. For a 75×75 physical problem size, run the code and report the runtimes and memory usage of both solvers.

Then, compute the estimated memory usage for a $10,000 \times 10,000$ physical problem size for both the "on-the-fly" sparse (where the LHS A matrix operator is never stored at all) and dense solver (where the A matrix operator is stored fully in dense format) techniques and report the values you come up with.

Additionally, plot the results for the 75×75 test problem with the default source term vector b found in our sample C code. Use a plotting script/program of your choice to generate the plot, and include the plot in your writeup.

For reference, we have included an animation that plots x after each iteration of CG, which is available at: `https://drive.google.com/open?id=1OMqZtL4uacsjrgIr7p_tJ_0V6LJMf698`. Note that as Equation 2 is a steady state equation, we only care about the final converged distribution, but an animation here is given in case you need to debug your code and want to see how it should look as the system converges.

3 MPI Parallelization (30 points)

In this section, your task will be to parallelize the optimized (sparse) "on-the-fly" version of your CG 2D Poisson solver. You should use MPI to implement a domain decomposition scheme. There are several considerations you will need to make when implementing your parallelization scheme:

- You will need to domain decompose more than one data structure. Specifically, the 1-D vectors x, b, r, p, z will all have to be decomposed across MPI ranks.
- While you are simulating a 2D problem, for the purposes of this assignment you are allowed to decompose in 1D. That is, if your 1D vectors x, b, r, p, z are of length N, and you are running on M MPI ranks, each rank will hold N/M contiguous elements of each 1D vector.
- While the pseudo-code for the CG solver found in Algorithm 1 remains largely the same when running in domain decomposed mode, you will need to implement parallel versions of your `axpy`, `dotp`, and "on-the-fly" `matvec` functions.
- `parallel_axpy` – The parallel version of this function should be very straightforward, as each subdomain of the vectors it operates on are independent. That is, no communication between MPI ranks is required to implement this function.
- `parallel_dotp` – The parallel version of this function requires some communication as a single scalar value is computed as the result. This function can be implemented by having all MPI ranks compute the dot product of their own local sub-domains, then performing an all-reduce across all MPI ranks to accumulate the overall dot product value.
- `parallel_matvec` – The parallel version of this function is the most complicated of the three. If you look at Equation 6 and the example serial code function `matvec_OTF`, you can see that for each index i in the vector w, you may need to access data from as far away as the $i - n$ and $i + n$ indices in w. As those indices may be outside of the local MPI sub-domain, you will need to use ghost cells and code up a ghost cell exchange routine between MPI ranks each iteration.
- You'll also need to code up a parallel function to populate the decomposed RHS b vector with the source term. Your source term function should be in the form of a small circle in the middle of the domain, as is defined in the example C code function `find_b`.

- We have provided empty functions for you to implement in the `parallel.c` file included with the example code. You are not required to use these functions, and may find that you need to add other functions or change functional interfaces. You are free to change the code or implement your own ideas in a different language, they are just provided for reference, there is no requirement to use/build off the example code.
- It is fine to assume that the size of the physical problem will be evenly divisible by the number of MPI ranks, just be sure to test and assert for this in your code.
- You should make it easy to switch between your serial and parallel solvers at runtime via a command line argument, in a similar fashion to what is used in the example C code we have provided.

4 MPI Parallel Plotting (10 points)

Write MPI code to output your solution x vector in matrix format for plotting your solution. Similar to previous problem sets, you are not allowed to have any single MPI rank hold the entire problem in its local memory. Besides that restriction, you are free to use whatever parallel file I/O methods you want in order to synchronize the serialization process. Be sure to test your parallel output file against the serial version, and include the resulting parallel plot in your writeup next to the serial plot version (they should be identical).

5 Performance Analysis

5.1 Strong Scaling (15 points)

Perform a strong scaling analysis of your code out to 64 MPI ranks using 4 nodes of the ”sandyb” partition of Midway. It's fine to run tests just using 1, 2, 4, 8, 16, 32, and 64 ranks. Record the runtimes you observe and create a plot of your strong scaling study in terms of speedup. Include your strong scaling plot in your PDF writeup.

For your tests, you'll need to determine a problem size that's reasonable for performance analysis. The problem size you pick should be large enough that that it takes at least 5-10 seconds when run on 64 ranks, though should not be so large that it takes prohibitively long to run the serial case.

5.2 Weak Scaling (15 points)

Perform a weak scaling analysis of your code out to 64 MPI ranks (using 4 nodes of the ”sandyb” partition of Midway). It's fine to run tests just using 1, 2, 4, 8, 16, 32, and 64 ranks. Record the runtimes, and create a plot of your weak scaling study. Include your weak scaling plot in your PDF writeup.

For your tests, you'll need to determine a problem size that's reasonable for performance analysis. The problem size you pick should run for at least 10 seconds when run in serial.

Does the resulting weak scaling plot look as expected given your experience in previous problem sets? If not, why not? Does the number of CG iterations required to converge the problem change with problem size?

5.3 Compiling, Code Cleanliness, and Documentation (10 pts)

We plan on compiling and running your code as part of grading it. You must include a makefile capable of compiling your code into a runnable executable. Also, while we are not grading to any rigid coding standard, your code should be human readable and well commented so we can understand what is going on.

5.4 What to Submit

You submission should be in the form of a zip or tar file containing the following items:

- Your source code. The serial and parallel versions can all be in the same repository in the form of different functions.

- A working makefile that compiles your code.
- Your plotting script(s)
- A single writeup (in PDF form) containing all plots and discussions asked for in the assignment