CS 240A Assignment 2: Solving Poisson's Equation by Conjugate Gradients

Assigned April 6, 2011

Due by 11:59 pm Monday, April 18

This assignment is to write a parallel program to implement the congugate gradient algorithm (CG for short), which solves a system of linear equations $Ax = b$ for any symmetric, positive definite matrix A; and to use your CG solver to solve a specific linear system called the "model problem", which is a discretization of Poisson's equation on a two-dimensional region.

We have supplied sequential Matlab versions of all of this code; your job will be to write a parallel MPI-based version. The main part of this will be to write three subroutines that do the bulk of the numerical work: `matvec` (which multiplies the model-problem matrix by a vector), `ddot` (which computes the inner product of two vectors), and `daxpy` (which scales and adds together two vectors).

You should do this assignment in a team of two students, coming from different departments. Exceptions must be approved in advance, and will be allowed only if we run out of candidates. Feel free to advertise on the course email list for a partner! After your team turns in its solution, your team will be paired randomly with another team: the two teams will exchange solutions, and you will write a review of the other team's solution and a comparison between them.

There's a lot of background for this problem in the slides from the April 6 lecture.

1 The model problem

Poisson's equation arises in many applications, including fluid dynamics, electromagnetics, and heat flow. Here we'll use it to compute the steady-state temperature in a square region whose boundary temperatures are known. The region is the unit square, $0 \le x \le 1$ and $0 \le y \le 1$. The temperature $t(x, y)$ satisfies Poisson's equation,

$$\frac{\partial^2 t(x,y)}{\partial x^2} + \frac{\partial^2 t(x,y)}{\partial y^2} = 0, \tag{1}$$

in the interior of the region, and the boundary the values of $t(x, y)$ are given as input.

To solve this problem numerically, we make two approximations: First, instead of asking for the answer at every point in the unit square region (an infinite problem!), we *discretize* the problem by filling the interior of the region with a finite, square grid of points. We 'll use k points in both the x and y directions, for $n = k^2$ points in all. (The choice of k will affect both how accurate and how expensive our solution will be.) We space the points evenly *inside* the unit square, so the distance between adjacent points is $1/(k + 1)$. Then we number the grid points from 1 to n, row by row, like this example with $k = 6$:

```
 1  2  3  4  5  6
 7  8  9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
31 32 33 34 35 36
```

The point in row r and column s of the grid has index $i = (r-1)k + s$; for example, the grid point in row 2, column 4 has index $i = (2-1) \cdot 6 + 4 = 10$. Now we will just try to compute the the n^2 temperatures $t(i)$ for each grid point i, instead of $t(x, y)$ for all real x and y.

Second, we approximate the derivatives in Poisson's equation by finite difference quotients. We know from calculus that

$$\frac{\partial^2 t(x,y)}{\partial x^2} = \lim_{h \to 0} \frac{t(x-h,y) - 2t(x,y) + t(x+h,y)}{h^2},$$

and

$$\frac{\partial^2 t(x,y)}{\partial y^2} = \lim_{h \to 0} \frac{t(x,y-h) - 2t(x,y) + t(x,y+h)}{h^2}.$$

We'll approximate the limits by taking h equal to the distance between grid points, which is $1/(k+1)$. (Here again the choice of k affects both the accuracy and the cost of our solution.) At almost every grid point i, then, the finite difference approximation to Poisson's equation (1) is

$$\frac{t(i-1) - 2t(i) + t(i+1)}{1/(k+1)^2} + \frac{t(i-k) - 2t(i) + t(i+k)}{1/(k+1)^2} = 0.$$

Here we're using the fact that the grid point i's neighbors on the left and right are points $i-1$ and $i+1$, and its neighbors above and below are points $i-k$ and $i+k$. Simplifying the equation a bit gives us

$$-t(i-k) - t(i-1) + 4t(i) - t(i+1) - t(i+k) = 0. \tag{2}$$

Equation (2) holds for every grid point i that's not next to a boundary of the region. If point i is next to the left boundary (for example), it has no left neighbor; instead, the temperature to its left is from the boundary conditions. Thus equation (2) becomes

$$-t(i-k) + 4t(i) + t(i+1) + t(i+k) = b(i),$$

where $b(i)$ is the boundary temperature next to grid point i. Similarly, the equations for the grid points next to the top, right, and bottom boundaries each have only three terms, and the equations for the four grid points at the corners are missing two terms each.

At the end of all this, we have one linear equation for each grid point i, which relates the unknown temperature $t(i)$ to at most four other unknowns, and to a right-hand side constant $b(i)$ that is either 0 or a given boundary temperature. These linear equations have a simple intuitive interpretation: They say that the temperature at every point is equal to the average of the temperatures at the surrounding points. This makes sense; if it weren't true at some point you'd expect the temperature there to be changing, but we assumed that the system is already in a steady state.

If we collect the coefficients of all n equations into a matrix A, and we let t be the vector whose elements are the n unknowns $t(1)$, $t(2)$, ..., and we let b be the vector whose elements are the right-hand size values $b(i)$, we can express the whole system of n equations in n unknowns as

$$At = b.$$

This is the so-called "model problem". Matrix A has a very special form. It is sparse, with at most 5 nonzeros in each row. All the diagonal elements are equal to 4, and all the nonzero off-diagonal elements are equal to -1. The matrix is symmetric, and it can be shown to be *positive definite*, which means that all its eigenvalues are positive.

2 The conjugate gradient algorithm

The conjugate gradient algorithm (CG for short) solves a system of linear equations, $Ax = b$, where the coefficient matrix A is symmetric and positive definite. CG is an iterative algorithm: It begins with a guess at x (usually the vector of all zeros) and improves it step by step. The theory of CG is quite pretty; there is an excellent description in the paper by Shewchuk linked to the course Resources page. For now, though, we will focus on what CG does rather than why it works.

There is a Matlab code for CG called `cgsolve.m` on the course web page. Here's a slightly simplified version of that code. The coefficient matrix is `A`; the right-hand side vector is `b`; variables `x`, `r`, `rnew`, and `d` are working vectors; and variables `alpha` and `beta` are scalars.

```
x = zeros(n,1);              % first guess is zero vector
r = b;                       % r = b - A*x starts equal to b
d = r;                       % first search direction is r
while (still iterating)
    alpha = r'*r / (d'*A*d);
    x = x + alpha * d;       % step to next guess
    rnew = r - alpha * A*d;  % update residual r
    beta = rnew'*rnew / r'*r;
    r = rnew;
    d = r + beta * d;        % compute new search direction
end;
```

One advantage of CG is that the only way it uses the matrix A is to multiply it by a vector. Indeed, the main loop only has a few kinds of algebraic operations: matrix-vector multiplication (called `matvec`); vector-vector inner product (called `ddot` for historical reasons); and scaling and adding two vectors (called `daxpy`). Therefore, one can use the same C code for CG in a wide variety of sequential and parallel settings, just by writing new subroutines for `matvec`, `ddot`, and `daxpy` each time. With a little care, you can implement CG so that during each iteration it just calls `matvec` once.

Deciding when to stop the iteration is a delicate question. Our Matlab code stops either after a fixed number of iterations, or when the size of the residual vector $r = b - Ax$ becomes small compared to the size of b. The speed of convergence of CG depends on the eigenvalues of A; for the model problem, one can prove that it converges to any specified degree of accuracy in a number of iterations that is proportional to $k = \sqrt{n}$.

3 Where's the data?

This is always the first question to ask about a parallel program, and in this case it boils down to two questions: how do you store the matrix and how do you store the vectors?

The answer to the first question might surprise you: You don't store the matrix A at all. Later in the course we'll look at parallel data structures for storing arbitrary sparse matrices, but the model problem's matrix is so simple that you don't even need to keep a copy of it!

Recall that the only way CG uses the matrix is in the `matvec` routine, which takes a vector as input and returns the matrix-vector product as output. We want to compute $v = Aw$ from w. Since we know exactly what A looks like, we can write down an explicit formula for each $v(i)$ in terms of the $w(i)$'s. For the interior grid points i, that formula is

$$v(i) = -w(i-k) - w(i-1) + 4w(i) - w(i+1) - w(i+k).$$

For grid points next to the boundary, one or more of the five w terms is missing. That's all there is to it. The Matlab code `modelmatvec.m` on the web site implements this sequentially. You'll need to do it in parallel.

How do you store the vectors? To use memory efficiently you'll distribute each vector's elements evenly, storing n/p elements of each vector on each processor. (For the homework assignment, let's suppose that that k is divisible by p, though you wouldn't write a library code that way.) The simplest way, and the first one you should implement, is to store elements 1 through n/p on processor 0, elements $n/p + 1$ through $2n/p$ on processor 1, and so on through processor $p - 1$. In terms of the grid of points in the unit square, this assigns the points in the first k/p rows to processor 0, the next k/p rows to processor 1, and so forth.

Inner products and vector addition are easy. Since

$$v^T w = \sum_{i=1}^{n} v(i)w(i),$$

`ddot` can compute the inner product in parallel by letting each processor compute and sum the products of its own elements of v and w, and then using communication to add all the processor sums together. Similarly, the `daxpy` operation can compute $v = \alpha v + \beta w$ by broadcasting α and β and then doing the vector addition in parallel.

How about `matvec`? Following the Matlab routine `modelmatvec.m`, if $v = Aw$ then each $v(i)$ depends on at most five elements of w, namely $w(i-k)$, $w(i-1)$, $w(i)$, $w(i+1)$, and $w(i+k)$. For most i, these five elements of w live on the same processor as $v(i)$, so no communication is necessary. In any given processor, $v(i)$ depends on data from a different processor exactly when grid point i is in the processor's first or last row of points. Thus `matvec` needs to send each processor's first and last row of grid point values (of w) to an adjacent processor.

There are two keys to doing the communication in `matvec` efficiently: First, don't broadcast all of w to every processor; rather, just send the parts of w that are needed, namely the values at the interprocessor boundaries. Second, send the interprocessor boundary values in a single message, not one at a time, to avoid paying the latency penalty (from the latency/bandwidth model) more often than necessary.

Exercise: What's the communication volume for one `matvec`?

4 What to implement

You will write the following C-MPI routines:

- `matvec`: As described above, this routine takes a distributed vector w and returns a distributed vector v. It does not store the matrix A. The sequential version is `modelmatvec.m` in Matlab.
- `ddot`: This routine takes two distributed vectors v and w (which might be the same) and returns the scalar dot product $\alpha = v^T w$.
- `daxpy`: This routine takes two distributed vectors v and w, and two scalars α and β, and overwrites vector v with $\alpha v + \beta w$.
- `getb`: This routine defines the right-hand side vector b. We've supplied one version of `getb()` in the harness, which corresponds to a solution whose correctess we can verify. You can modify `getb()` to represent other boundary conditions if you'd like.
- `cgsolve`: This is the actual CG solver. The sequential version is `cgsolve.m` in Matlab. This routine should not contain anything specific to the model problem—only `matvec()` and `getb()` know what A and b are.

5 Test harnesses

A framework for you to use in implementing and testing your routines is linked to the course web page. This includes a skeleton `main()` program with input and output, a `Makefile`, and an example of a `getb()` routine that gives values of b for one particular set of boundary conditions. Note that `getb()` returns one element at a time: the call `bi = getb(i,n)` returns $b(i)$ (here n is the dimension of the matrix). You should use `getb()` at the beginning of your program (not inside the CG loop) to set up the right-hand side vector in parallel; each processor can call `getb()` to get its own elements of b. The framework also has a verification routine that checks the answer for our example `getb()`.

6 What experiments to do

As always, you should first debug your code with very small values of k, on one processor, then two processors, then several processors. For debugging you should probably only do a couple of iterations of CG. You should compare your results for very small problems to the results from the Matlab code on the web site.

Here are some experiments to do:

1. (Strong scaling analysis.) Choose a value of k for which your code runs on one processor in a reasonable amount of time, say 30 seconds to a minute. Run your code for this k with $p = 1$, 4, 8, 12, 16, and 32. For each run, report the running time and the parallel efficiency. Make plots of the running time versus p, and of the parallel efficiency versus p.
2. (Weak scaling analysis.) Change your program to do only 100 iterations of the CG loop, to make the experiments in this part run faster. Now choose a starting value of k to use for $p = 1$, possibly the same k as above. Run your code for $p = 1$, 4, 8, 12, 16, and 32, but this time use a different k for each p, chosen so that k is (nearly) proportional to $\sqrt{p}$. Since both total memory and total work scale as $n = k^2$, this implies that the memory required per processor and the work done per processor will remain constant as you increase p. This is called *weak scaling*. Again, report the running time and the parallel efficiency for each run, and make the same plots you did for the previous experiment.

3. Change your program to do only 10 iterations of the main CG loop, and then do the most detailed analysis possible with the Triton performance tools (TAU and paraprof or pprof). Make plots of the results.

Report on your experiments and your results. What trends do you see? Do the running time and efficiency behave as you would expect? Can you explain your results? (Warning: Experimental timing results on parallel codes are often not nearly as clean as you might expect from the theory!)

7 Extra credit

Here are some ideas for extensions to the problem. I particularly recommend the first one.

1. **Two-dimensional data decomposition.** Try a different answer to the "where's the (vector) data" question: Instead of assigning the k/p rows of grid points to each processor, divide the grid into p square regions of $k/\sqrt{p}$ by $k/\sqrt{p}$ points each, one per processor. (It's ok to take p as a perfect square.) Theoretically, how does this affect the communication volume? Experimentally, do your results support the theory or not?

2. **Ghost nodes.** This is an idea that trades off a little extra memory and computation for a reduction in the number of messages sent between processors. It sends fewer, bigger messages. It doesn't reduce the communication volume, but in the latency/bandwidth model it reduces the α term, which can be important. The idea is that, instead of sending one layer of interprocessor boundary grid values at a time (in `matvec`), you send 2, 3, or some (small) number l of layers at once. If a processor receives l layers of boundary nodes from its neighbors, it can then do l CG iterations without further communication in the `matvec`. The tradeoff is that the processor needs memory to store the extra layers of "ghost nodes", and also there is some extra computation because the ghost values are being computed redundantly by two adjacent processors. Experiment to find the best number of ghost layers.

3. **Other matrices.** By changing only the `matvec()` and `getb()` routines, you can use your CG code to solve any sparse linear system with a positive definite coefficient matrix. (You may need to store the matrix explicitly, in parallel.) Download some large symmetric positive definite matrices from the Florida Sparse Matrix Collection (see Google) and try out your code on them.

4. **Comparison with Jacobi's method.** Jacobi is an iterative method that is easier to describe than CG but converges more slowly. The idea is simple: An iteration consists of changing the value of $x(i)$ so that equation number i is satisfied exactly, simultaneously for all i. In the model problem, this is the same as setting the next guessed temperature at a grid point to be the average of the current guessed temperatures at the four neighbors, simultaneously for all grid points. You can write Jacobi as a matrix iteration too. Let $D = 4I$ be the diagonal part of A. Then the Jacobi iteration is

$$x^{(k)} = D^{-1}(b - (A - D)x^{(k-1)}).$$

8 Reviewing another team's code

When you turn in your report and your code on Monday, April 18, we will pair you randomly with another team. Before Friday, April 22, you will read their report, run their code (if you can),

and write a one-page review report giving your observations and discussing the similarities and differences between their code and yours. Of course, the other team is meanwhile doing the same with your code and report. You'll turn in your review report on Friday, April 22.