Programming Assignment 2: The Conjugate Gradient Method

Parallel and Distributed Computing

Due Wednesday, February 26
Note the change in the due date

1 The Method of Conjugate Gradients

Solving linear systems is one of the most important problems in scientific computing. In most cases of interest the coefficient matrix is sparse, i.e., the number of zeroes is so large that storing the matrix as a two-dimensional array is impractical or impossible. Unfortunately, the *direct methods* that you learned in elementary linear algebra classes (e.g., Gaussian elimination) are very difficult to adapt for distributed memory systems when the coefficient matrix is sparse. So many of the sparse linear system solvers for distributed memory systems use *iterative methods.* An iterative method makes an initial guess at a solution to the system, and then tries to repeatedly improve the guess. Thus, iterative methods produce a sequence of vectors, $x_0, x_1, \ldots$ each of which is an approximation to the solution to the linear system. For programming assignment 2, you should use C and MPI to implement the iterative method known as the *method of conjugate gradients.*

The method of conjugate gradients is used to solve linear systems when the coefficient matrix is *symmetric* and *positive definite.* An $n \times n$ matrix

$$A = [a_{ij}] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}.$$

is *symmetric* if $a_{ij} = a_{ji}$ for $i, j = 1, \ldots, n$. The matrix is *positive definite* if for every nonzero vector x, the scalar

$$x \cdot Ax > 0.$$

(Here $\cdot$ denotes the dot product of the two column vectors x and Ax.)

The method of conjugate gradients can also be viewed as solving a minimization problem: Suppose

$$q(x) = \frac{1}{2} x \cdot Ax - x \cdot b,$$

where A is a symmetric positive definite $n \times n$ matrix, x and b are n-dimensional column vectors, and $\cdot$ denotes the dot product. In order to find the minimum value of q, we can

compute its gradient which turns out to be

$$\nabla q(x) = Ax - b,$$

and it can be shown that the solution to $Ax - b = 0$ is the unique minimum of q. So minimizing q and solving the linear system $Ax = b$ are equivalent problems.

After making an initial guess at a solution to the linear equation (or the minimum of q), the conjugate gradient method proceeds by choosing "search directions"

$$p_1, p_2, \ldots$$

Each of these directions is just an n-dimensional vector. Once a direction is chosen, a new estimate for the solution is obtained by minimizing q along the new direction. That is, if p_k is the new search direction, and x_{k-1} is the previous estimate of a solution, then the new estimate, x_k, will be chosen by choosing the scalar α_k that minimizes

$$\min_{\alpha} q(x_{k-1} + \alpha p_k).$$

So we need to see how to choose the search directions. A key ingredient in the choice is the *residual.* If x_k is one of our estimates, then the corresponding residual is

$$r_k = b - Ax_k.$$

It can be viewed as a measure of how close x_k is to the solution to the original linear system. Having chosen search directions $p_1, p_2, \ldots, p_{k-1}$, and computed estimates $x_0, x_1, \ldots, x_{k-1}$ and residuals $r_0, r_1, \ldots, r_{k-1}$, the method chooses the new search direction, p_k, from the orthogonal complement of

$$\text{Span}\{Ap_1, Ap_2, \ldots, Ap_{k-1}\}.$$

More explicitly, p_k is chosen from this subspace so that it minimizes the distance

$$\|p_k - r_{k-1}\|$$

in the usual Euclidean norm. It can be shown that if

$$\beta_k = \frac{r_{k-1} \cdot r_{k-1}}{r_{k-2} \cdot r_{k-2}},$$

then p_k will minimize $\|p_k - r_{k-1}\|$, when

$$p_k = r_{k-1} + \beta_k p_{k-1}.$$

With this choice of p_k, it can also be shown that if

$$\alpha_k = \frac{r_{k-1} \cdot r_{k-1}}{p_k \cdot Ap_k},$$

then

$$x_k = x_{k-1} + \alpha_k p_k$$

will minimize $q(x_{k-1} + \alpha p_k)$.

So we can write the conjugate gradient method as follows:

$k = 0$; $x_0 = 0$; $r_0 = b$
while ($\|r_k\|^2 >$ tolerance) **and** ($k <$ max_iter)
    $k++$
    if $k = 1$
        $p_1 = r_0$
    else
        $\beta_k = \frac{r_{k-1} \cdot r_{k-1}}{r_{k-2} \cdot r_{k-2}}$ // minimize $\|p_k - r_{k-1}\|$
        $p_k = r_{k-1} + \beta_k p_{k-1}$
    endif
    $s_k = Ap_k$
    $\alpha_k = \frac{r_{k-1} \cdot r_{k-1}}{p_k \cdot s_k}$ // minimize $q(x_{k-1} + \alpha p_k)$
    $x_k = x_{k-1} + \alpha_k p_k$
    $r_k = r_{k-1} - \alpha_k s_k$
endwhile
$x = x_k$

The calculation of r_k is just a shortcut to save a matrix-vector multiplication:

$$r_k = b - Ax_k = b - A(x_{k-1} + \alpha_k p_k) = r_{k-1} - \alpha_k s_k.$$

Further details on the algorithm can be found in many books on scientific computing and numerical analysis. A classic in the field is *Matrix Computations* by Gene Golub and Charles van Loan. A nice discussion can be found in the Wikipedia article "Conjugate Gradient Method."

Note that the algorithm terminates when the square of the norm of the residual is less than some user-specified tolerance or when the number of iterations has exceeded a user-specified maximum. In the first case, the method has converged, while in the second case, the method has failed.

As noted earlier, the residual can be viewed as a measure of how close the iterate is to the solution. However, it should *not* be confused with the error: the error in the kth iterate is the difference between the exact solution and the iterate:

$$e_k = A^{-1}b - x_k.$$

Thus, in exact arithmetic the residual is related to the error by the formula

$$r_k = Ae_k,$$

and there may be a large difference between the two. In practice, however, the error is usually unavailable — if you knew the error and the iterate, you'd know the solution $A^{-1}b$. So most iterative methods rely on the residual for determining when to stop.

It should also be noted that in exact arithmetic, the conjugate gradient method converges to the solution in n iterations. However, because of rounding errors, this won't, in general, be the case on the computer. In fact, for the very large systems solved on distributed memory machines — systems with hundreds of thousands or millions of unknowns — n iterations would take far too long.

Finally, we should note that in practice the method of conjugate gradients is usually too slow to converge. So commercial packages using conjugate gradient solvers typically solve a *preconditioned* linear system. Such a system is often obtained from the original system by matrix multiplication:

$$M^{-1}Ax = M^{-1}b$$

Here M is an $n \times n$ matrix that is chosen so that the iterative method will converge in substantially fewer iterations than the original system. For further information, see Golub and Van Loan.

2 Programming Assignment

For assignment 2, you should write a double precision, parallel conjugate gradient solver for distributed memory systems using C and MPI. You do *not* have to implement preconditioned conjugate gradients.

The input to the program will consist of two parts: command line options and input to `stdin`. The command line options will be, in order,

- The order of the system,
- The tolerance,
- The maximum number of iterations, and
- An optional fourth argument that will suppress output of the solution vector.

Thus, the program might be started with a command-line that looks something like this:

```
$ mpiexec -n 4 ./conj_grad 100 1.0e-6 50
```

If the user wants to suppress output of the solution vector, she should add the character 'n' to the command line:

```
$ mpiexec -n 4 ./conj_grad 100 1.0e-6 50 n
```

The input to `stdin` will consist of, in order,

- The matrix A, and
- The right-hand side b.

This will consist of $n(n+1)$ doubles (and white space). Of course, for large matrices and vectors, the input can be redirected from a file:

```
$ mpiexec -n 4 ./conj_grad 100 1.0e-6 50 < big_file
```

or

```
$ mpiexec -n 4 ./conj_grad 100 1.0e-6 50 n < big_file
```

The output should be printed to `stdout`, and it should include

- The number of iterations,
- The time used by the solver (not including I/O),
- The solution to the linear system,
- The norm of the residual calculated by the conjugate gradient method, and
- The norm of the residual calculated directly from the definition of residual.

However, as we noted above, you should also allow the user an option to suppress output of the solution.

All of the input data will be entered on process 0 and all of the output should be printed by process 0. The command-line arguments should be parsed by process 0 and broadcast to the other processes.

It will probably be easiest to use a block-row distribution of the matrix and block distributions of the vectors. However, if you prefer, you can use a block-column distribution of the matrix. In either case you can assume that the order of the matrix n is evenly divisible by the number of processes. You can also use any MPI functions in your program.

Note that you do *not* have to make use of special sparse matrix storage methods. You can store your matrix data structure in any way that's convenient. As we noted for programming assignment 1 most C programs store matrices as 1-dimensional arrays, and do the subscripting by "hands." For example, if `A` stores an $n \times n$ matrix as a 1-dimensional array, then we can access the element in the ith row and jth column with `A[i*n+j]`.

Also note that you should *not* store all of the vectors r_k, p_k, s_k, and x_k: this would require a *huge* amount of storage for a large problem. You should only store what's needed. In fact, you only need to store the "current" copy of each of these vectors: after the $(k-1)$st iterate is used to initialize the kth iterate, the $(k-1)$st iterate isn't used again.

3 Program Development

For this program, you may find it helpful to proceed as follows.

1. Write a serial conjugate gradient program. The program should allow some flexibility with input: it should allow you to directly enter the system and it should also generate systems automatically given their order. You can generate symmetric positive definite matrices by generating a nonsingular square matrix C using a random number generator. Don't use the generator `rand`. It does such a poor job of generating numbers that matrices of order ten can be singular. A better generator is `random` in `stdlib`. After generating a random nonsingular matrix C, you can get a symmetric positive definite matrix A by multiplying C by its transpose: $A = CC^T$. You can generate exact solutions by simply generating a random solution x and computing the right-hand side b by multiplying Ax.

2. Once you've written the serial program, write the I/O functions for the parallel program. Be sure to get these functions fully debugged before starting to work on the rest of the program — the information you get from these functions will be crucial during development.

3. Write a parallel dot product, a parallel daxpy, a parallel matrix-vector product, and a parallel scalar-vector product. Daxpy is just the double-precision version of saxpy ("*s*calar α x *p*lus y"):

$$y = y + \alpha x.$$

4. Once you've got these functions fully debugged, it shouldn't be too difficult to implement the parallel conjugate gradient solver.

5. You should be able to significantly improve the performance of your solver by optimizing the basic linear algebra functions.

4 Documentation

Follow the standard rules for documentation. Describe the purpose and algorithm, the parameters and variables, and the input and output of each routine.

5 Grading

Correctness will be 75% of your grade. Does your program correctly solve linear systems using the conjugate gradient method? Does it accept input and command line options in the required format? Does it output the correct information?

Quality of solution will be 20% of your grade. Is the implementation reasonably efficient? Is the program well-designed and easy-to-read?

Static features such as source format and documentation will be 5% of your grade.

Contest After the programs are submitted, I'll run a contest. The fastest solver will receive 10 points extra credit. The second and third fastest solvers will each receive 5 points extra credit. Note however, only correct, well-designed programs are eligible.

6 Collaboration

You may *discuss* all aspects of the program with your classmates. However, you should never show any of your code to another student, and you should not look at anyone else's code — regardless of its source.

7 Submission

The program is due on *Wednesday, February 26.* By 11 am, you should have copied your source files, makefiles, and any documentation to your svn directory `cs625/prog2`. Put a hardcopy of your source code in my mailbox in Harney 545 by 2 pm.