CS 240A Assignment 2: Matrix-Vector Multiplication and the Power Method

Assigned April 5, 2010

Due by 11:59 pm Monday, April 12

This assignment is to write a parallel program to multiply a matrix by a vector, and to use this routine in an implementation of the power method to find the absolute value of the largest eigenvalue of the matrix.

Your code will call routines that we supply to generate matrices, record timings, and validate the answer.

You may do this assigment in teams of two students. (Be sure to put both names on your submission.)

1 Mathematical background

A square *matrix* is an n-by-n array A of numbers. The entry in row i, column j of A is written either a_{ij} or $A(i, j)$. The rows and columns are numbered from 1 to n. A *vector* is a one-dimensional array x whose i'th entry is x_i or $x(i)$. Recall the definition of matrix-vector multiplication: The product $y = Ax$ is a vector y whose elements are

$$y_i = \sum_{j=1}^{n} a_{ij} x_j .$$

In words, each element of y is obtained from one row of A and all of x, by computing an inner product (that is, by adding up the pointwise products). Every element of x contributes to every element of y; each element of A is used exactly once.

The *power method* uses matrix-vector multiplication to estimate the size of the largest eigenvalue of a matrix A, which is also called the *spectral radius* of A. It works as follows. Start with an arbitrary vector x. Then repeat the following two steps: divide each element of x by the length (or *norm*) of x; second, replace x by the matrix-vector product Ax. The length of the vector eventually converges to the spectral radius of A. In your code, you will repeat the matrix-vector product 1000 times.

There is a sequential Matlab code for the power method on the course web page (under Homework 2). You will write a parallel C/MPI code that does the same computation.

2 Test harnesses

We will supply some routines that you can call from your code for testing. We will also use these routines in grading your program. The routines include a matrix generator, a validator that checks correctness of the results (for the particular matrix we generate), and a timer. See the FAQ on the course web page for more information on the test harnesses.

3 What to implement

You will write the following C routines:

- `generateMatrix`: Although we'll supply a matrix generator for grading, you should also generate various matrices for testing your code. This routine generates a matrix of specified size, with the data distributed across the processors as specified in the next section.
- `powerMethod`: Implements the power method on a given matrix (which is already distributed across the processors). This routine calls `norm` and `matVec`.
- `norm`: Computes the norm (the length) of a given vector.
- `matVec`: Multiplies a given matrix (which is already distributed across the processors) by a given vector.
- `main`: The main routine that calls `generateMatrix` and `powerMethod`, and times `powerMethod`.

4 Where's the data?

You may assume that n, the number of rows and columns of the matrix, is divisible by p, the number of processors. Distribute the matrix across the processors by rows, with the same number of rows on each processor; thus, processor 0 gets rows 1 through n/p of A, processor 1 gets rows $n/p + 1$ through $2n/p$, and so forth. Your `generateMatrix` routine should not do any communication except for the value of n; each processor should generate its own rows of the matrix, independently of the others, in parallel.

Put the vector on processor 0. For our purposes, the "arbitrary vector" you start with can be a vector of random doubles.

When you write your `matVec` routine, you should do the communication with `MPI_Bcast` and `MPI_Gather`; you will find the code to be much simpler this way than if you do it all with `MPI_Send` and `MPI_Recv`.

5 What experiments to do

First, debug your code on very small matrices, on one processor, then two processors, then several processors. Two matrices you can use for debugging are the n-by-n matrix of all ones (whose spectral radius is n) and the identity matrix (with ones on the main diagonal and zeros elsewhere), whose spectral radius is 1. For debugging you should probably only do a few iterations of the power method instead of 1000.

Your code should only time the call to `powerMethod`, not the matrix generation. You should call our harness routines to start and stop the "official" timer (see the FAQ); you can also use `MPI_Wtime` for your own timings.

Here are some experiments to do:

1. (Strong scaling analysis.) Choose a value of n for which your code runs on one processor in a reasonable amount of time, say 30 seconds to a minute, with 1000 iterations of the main loop. (On my old one-processor laptop, $n = 3000$ takes about 40 seconds.) Run your code on this matrix for $p = 1, 4, 8, 12, 16$, and (if possible) 32. For each run, report the running time and the parallel efficiency. Make plots of the running time versus p, and the parallel efficiency versus p.

2. (Weak scaling analysis.) Change your program to do only 100 iterations of the main loop, to make the experiments in this part run faster. Now choose a starting value of n to use for $p = 1$, possibly the same n as above. Run your code for $p = 1$, 4, 8, 12, 16, and 32, but this time use a different n for each p, chosen so that n is (nearly) proportional to $\sqrt{p}$. Since both total memory and total work scale as n^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. (Performance tools.) At press time, the state of the performance tools on Triton was still unclear; watch the FAQ for updates. Assuming we get the tools working, for this experiment you should change your program to do only 10 iterations of the main loop, and then do the most detailed analysis possible with the Triton performance tools. 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!)

6 Extra credit

For extra credit, you may experiment with different data layouts. If the matrix is distributed by columns instead of by rows, the `matVec` routine will use `MPI_Scatter` and `MPI_Reduce`. You can also try distributing the matrix by blocks. In that case, you may want to use separate MPI communicators for the rows and columns of processors. One way to do this is with `MPI_Comm_split`, as described in chapter 5 of the Pacheco MPI tutorial.