CS 140 Assignment 2:
Matrix-Vector Multiplication and the Power Method

Assigned January 11, 2010

Due by 11:59 pm Wednesday, January 20

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. You will write separate functions to generate the matrix and to perform the power method, and you will do some timing experiments with the power method routine.

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 What to implement

You will write the following C routines:

- `generateMatrix`: Generates a matrix of specified size, with the data distributed across the processors as specified in the next section. The section on experiments below defines the matrices you'll use for experiments.
- `powerMethod`: Implements the power method on a given matrix (which is already distributed across the processors). This routine calls `norm2` and `matVec`.

- `norm2`: 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 does the timing.

3 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 should be the vector whose elements are all equal to 1.

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`.

4 What experiments to do

When you debug parallel code, you have to go very slowly and take one step at a time; otherwise, when something breaks it is very hard to figure out where the problem is. Use lots of `printf()` statements!

First, debug your `matVec` routine by itself (called from `main`), first on one processor, then two processors, then several processors. Using very small matrices, check to make sure your routine is getting the same answers as the Matlab code. (Hint: In Matlab, if you omit the semicolon at the end of any statement, it prints the value computed by the statement; try running `powermethod.m` with and without semicolons inside the main loop.)

Second, debug the whole code, always starting with tiny matrices on one, then two, then four processors. For debugging you will only have to do two or three iterations of the power method to see if you're getting the same results as Matlab.

The Matlab code `generateMatrix.m` generates a specific matrix (for which you supply the size n) that you can use for both debugging and timing experiments. Here is the matrix for $n=6$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 & 0 & 0 \\ 3 & 3 & 3 & 0 & 0 & 0 \\ 4 & 4 & 4 & 4 & 0 & 0 \\ 5 & 5 & 5 & 5 & 5 & 0 \\ 6 & 6 & 6 & 6 & 6 & 6 \end{pmatrix}$$

Because this matrix is triangular (has only zeros above the main diagonal), its eigenvalues are equal to its diagonal elements, and the spectral radius is therefore equal to n, the largest diagonal element. However, the power method is pretty slow to converge—for very large n you won't get an accurate estimate of the spectral radius in 1000 iterations. That's okay; you can check your accuracy on small matrices and use big ones just for timing experiments.

Your code should use `MPI_Wtime`, and for your timing experiments it should only time the call to `powerMethod`, not the matrix generation.

Here are the experiments you should do:

1. 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 one-processor laptop, $n = 3000$ takes about 40 seconds.) Run your code on this matrix for $p = 1$, 4, 8, and 16. For each run, report the running time t_p and the parallel efficiency $t_1/(pt_p)$. Make plots of the running time versus p, and the parallel efficiency versus p. (You can use Matlab to plot the data, or any other plot package you want.) You will want to wait to run on 8 and 16 processors until you are absolutely sure your code is giving the right answers on 2 and 4 processors.

2. Change your program to do only 10 iterations of the main loop, and recompile it with a profiling tool. Run a profile of your code, for $p = 4$ and $p = 16$. Note that the profiling tools weren't working yet as of Monday; we'll send a note to the Google group later this week with more details about what kind of profiles we want you to do.

Write a report that describes 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!)

You can debug your code on any machine you like (for example, you can use CSIL for debugging with $p = 1$ and $p = 2$), but the results you turn in must come from runs on Triton.

5 What to turn in

Use `ssh` or `scp` to copy your files from Triton to CSIL, and then use `turnin hw2@cs140` from CSIL to turn in the following:

- Your source code.
- A `Makefile` that compiles your code.
- A report (as a text file or PDF, named `report.txt` or `report.pdf`, respectively) containing instructions for compiling and running your code, plus tables of your run time results, plus your plots, plus a description and interpretation of your results and any conclusions you draw from them.
- Your directory structure should look like:

  `./`

  `./Makefile`

  `./report.pdf` or `./report.txt`

  `./your-source-files`

  and so on with whatever files you need.
- If you worked in a team, your report should include the names and CSIL accounts (or perm number if you don't have an account) of both members.
- Don't turn in any executable or .o files.

You may do this assignment alone or in groups of two.