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

Assigned January 13, 2009

Due by 11:59 pm Thursday, January 22

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. For extra credit, you may experiment with different MPI routines and with different ways of laying out the data.

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 `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`.

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

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 use `MPI_Wtime`, and it should only time the call to `powerMethod`, not the matrix generation.

For your timing experiments you'll use an n-by-n matrix that has the value n on the main diagonal and also on the diagonals above and below the main diagonal, and zeros everywhere else. Here is the matrix for $n = 6$:

$$\begin{pmatrix} 6 & 6 & 0 & 0 & 0 & 0 \\ 6 & 6 & 6 & 0 & 0 & 0 \\ 0 & 6 & 6 & 6 & 0 & 0 \\ 0 & 0 & 6 & 6 & 6 & 0 \\ 0 & 0 & 0 & 6 & 6 & 6 \\ 0 & 0 & 0 & 0 & 6 & 6 \end{pmatrix}$$

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, 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. Change your program to do only 10 iterations of the main loop, and recompile it with VAMPIR. Use VAMPIR to make plots of the global timeline display as Stefan B. showed in class, for $p = 4$ and $p = 16$, rather like the one at

   `http://www.nersc.gov/nusers/resources/software/tools/vampir.gif`

   Turn in the VAMPIR plots in any standard format so that we can see them. You may want to try out other kinds of VAMPIR plots as well.

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!)

5 Extra credit

There are two ways to get extra credit on this problem:

1. Write a second version of `matVec` that uses only `MPI_Send` and `MPI_Recv` instead of `MPI_Bcast` and `MPI_Gather`. Compare the performance of the two versions, and comment on your results.

2. Write a second version of your entire program that distributes the matrix across the processors by columns instead of by rows. For this version, use `MPI_Scatter` and `MPI_Reduce` for the communication in `matVec`. How do your results compare to those of the row-wise layout?

6 What to turn in

Use `ssh` or `scp` to copy your files from TeraGrid 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, word file, or pdf) 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.
- If you worked in a team, your report should include the names and perm numbers of both members.
- Don't turn in any executable or .o files.

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