Lab

POSIX Threads

The purpose of this lab is for us to perform a standard computational task using threads, which will help ease us into the threading component of the multithreading programming assignment. In particular, we will get our hands dirty with `pthreads` by writing our own routine for doing *matrix-vector multiplication*. This task is an essential subroutine of many machine learning and data science algorithms.

Let A be a $m \times n$ matrix and let x be a $n \times 1$ column vector.

$$A = \begin{pmatrix} a_{1,1}, a_{1,2}, \cdots, a_{1,n} \\ a_{2,1}, a_{2,2}, \cdots, a_{2,n} \\ \vdots, \vdots, \cdots, \vdots \\ a_{m,1}, a_{m,2}, \cdots, a_{m,n} \end{pmatrix}, \qquad x = (x_1, x_2, \cdots, x_n)^\top.$$

The multiplication Ax gives a $m \times 1$ column vector y defined as follows:

$$Ax = \begin{pmatrix} \sum_{j=1}^n a_{1,j} x_j \\ \sum_{j=1}^n a_{2,j} x_j \\ \vdots \\ \sum_{j=1}^n a_{m,j} x_j \end{pmatrix} = y.$$

Serial Implementation

In serial (i.e., without threads) build a $m \times n$ matrix A filled with random numbers, and a $n \times 1$ vector x filled with random numbers. Then write a serial matrix-vector multiplication routine that returns $Ax = y$.

Multithreaded Implementation

Use `pthreads` to perform the matrix-vector multiplication $Ax = y$. The number of threads to be used will be specified by the user, i.e., it will be in `argv[1]`. At first, you may assume that the number of threads equals m, the number of rows in your matrix A, but then you should write your program so that it balances the work to be done among the threads when the number of threads is less than the number of rows (such load balancing is needed for the multithreading programming assignment).

When you finish, race your serial solution against your multithreaded solution for a large matrix. The larger you make your matrix, the more your multithreaded version should vastly outperform your serial implementation. For very large matrices, you will need to be careful about your implementation, i.e., watch out for integer overflow.

To begin, download the provided `C` source code and use that as a skeleton. The skeleton code may not compile, so use this as a template to fill in the details of your implementation. If you get a segfault, you are probably not passing command-line arguments properly. It might be useful to review sample code that already shows some basic written examples of how to use `pthreads`. Don't forget when using `pthreads`, we need to pass the `-lpthread` flag to `gcc` and `-lm` if you are using the math library.