Matrix-vector multiplication

What is Matrix-Vector multiplication?

Let's take a look at writing a Pthreads matrix-vector multiplication program. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})^T$ is an n-dimensional column vector,[4] then the matrix-vector product $A\mathbf{x} = \mathbf{y}$ is an m-dimensional column vector, $\mathbf{y} = (y_0, y_1, \ldots, y_{m-1})^T$, in which the ith component y_i is obtained by finding the dot product of the ith row of A with $\mathbf{x}$:

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j.$$

- You are essentially taking the value of the matrix represented as a_{ij} and the vector as x_j and multiplying them together and then summing all of them up

Difference between matrix and vector:

- Matrix: is a an array of m rows x n columns
- Vector: is an array of 1 row x n columns or 1 columns x n rows
- In matrix vector multiplication you need the amount of columns in the matrix to match up with the number of rows in vector

What it looks like:

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

- Matrix on the left and vector in the middle, which gives you a y for each completion of a row in the matrix times the vector.

Pseudocode:

Thus pseudocode for a *serial* program for matrix-vector multiplication might look like this:

```
/* For each row of A */
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   /* For each element of the row and each element of x */
   for (j = 0; j < n; j++)
      y[i] += A[i][j]* x[j];
}
```

We want to parallelize this by dividing the work among the threads.

- One way to do this is to have each thread handle an even amount of rows from the matrix and have each thread compute the sum of the rows it handles

We want to parallelize this by dividing the work among the threads. One possibility is to divide the iterations of the outer loop among the threads. If we do this, each thread will compute some of the components of `y`. For example, suppose that $m = n = 6$ and the number of threads, `thread_count` or t, is three. Then the computation could be divided among the threads as follows:

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

To compute `y[0]`, thread 0 will need to execute the code

```
y[0] = 0.0;
for (j = 0; j < n; j++)
   y[0] += A[0][j]* x[j];
```

Therefore thread 0 will need to access every element of row 0 of `A` and every element of `x`. More generally, the thread that has been assigned `y[i]` will need to execute the code

```
y[i] = 0.0;
for (j = 0; j < n; j++)
   y[i] += A[i][j]*x[j];
```

Thus this thread will need to access every element of row `i` of `A` and every element of `x`. We see that each thread needs to access every component of `x`, while each thread only needs to access its assigned rows of `A` and assigned components of `y`. This suggests that, at a minimum, `x` should be shared. Let's also make `A` and `y` shared. This might seem to violate our principle that we should only make variables global that need to be global. However, in the exercises, we'll take a closer look at some of the issues involved in making the `A` and `y` variables local to the thread function, and we'll see that making them global can make good sense. At this point, we'll just observe that if they are global, the main thread can easily initialize all of `A` by just reading its entries from `stdin`, and the product vector `y` can be easily printed by the main thread.

- Code for the thread function

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
          y[i] += A[i][j]*x[j];
   }

   return NULL;
}  /* Pth_mat_vect */
```

Program 4.2: Pthreads matrix-vector multiplication.

  - Here we are going to assume that..
    at a minimum, x should be shared. Let's also make A and y shared. This might seem to violate our principle that we should only make variables global that need to be global. However, in the exercises, we'll take a closer look at some of the issues involved in making the A and y variables local to the thread function, and we'll see that making them global can make good sense. At this point, we'll just observe that if they are global, the main thread can easily initialize all of A by just reading its entries from stdin, and the product vector y can be easily printed by the main thread.
- Some programs are easier to write with threads than with processes due to shared memory…

If you have already read the MPI chapter, you may recall that it took more work to write a matrix-vector multiplication program using MPI. This was because of the fact that the data structures were necessarily distributed, that is, each MPI process only has direct access to its own local memory. Thus for the MPI code, we need to explicitly *gather* all of `x` into each process's memory. We see from this example that there are instances in which writing shared-memory programs is easier than writing distributed-memory programs. However, we'll shortly see that there are situations in which shared-memory programs can be more complex.