Goal: Modify the serial program given in file serial_vector_rotate.c to parallelize it using Pthreads.

There are two input files provided for testing the program: input1.txt and input2.txt. It is not necessary to understand the mathematics in the program, since you're only modifying the control flow. You can verify that your program does not change the mathematics by checking the results on the two data sets.

Those results should be:

```
Result = [-613.67, 28.55, -162.24]        Result = [-661.51, -234.53, -179.22]
```

For input1 and input2 respectively.

Programming Requirements

For this assignment, it is sufficient to parallelize the two loops that operate over the array of vectors. (This code is clearly marked in the `main()` function.) You do not need to attempt to parallelize the input function or the computation of the rotation matrix.

The first loop multiplies the input vectors by the rotation matrix and stores the results in the output vector array:

```
for (v=0; v<num_vectors; v++)
        multMatrixVector(rotation_matrix, &(input_vectors[v*3]),
                                          &(output_vectors[v*3]));
```

Each pass through this loop is independent, so it is possible to compute all multiplications simultaneously.
The second loop sums the vectors in the output vector array:

```
for (v=0; v<num_vectors; v++)
{
    addVectorVector(result, &(output_vectors[v*3]), temp);
    result[0] = temp[0]; result[1] = temp[1]; result[2] = temp[2];
}
```

The summation does have dependencies because each addition cannot simultaneously access the result vector. When you parallelize this loop, have each thread do as much computation as possible using local variables. Each thread should make exactly one write to the result vector, and that write must be in a protected critical section.

Your program should be written to allow for a variable number of threads and a variable number of vectors. You can assume the number of threads and the number of vectors are both powers of 2. The number of vectors is determined by the input file, and is already coded into the program. The number of threads should be determined by a command line argument, which you will need to add to the program.

The computational work should be evenly distributed among the available threads.

Collecting Timing Data

Reading the input data file takes substantially more time than the computational work. You should not include the time consumed but the `readInputDatafile()` function in any of your timing analysis.
You need to use timer.h (Check the comments for proper usage of GET_TIME).
Use a Barrier to properly time the appropriate section.

Note: While it may seem unrealistic to ignore the data read time, there are scenarios in which that time could be reduced or ignored. The read time could be substantially reduced by converting the data files to a binary format, but that is out of scope of this assignment.

Tasks

Task 1: Design and implement the parallel version of this program, following the programming requirements given above.

Task 2: Record timing data for this program for a sequence of different thread counts, using the larger input file `input2.txt`. You should time the program for 2n threads, where n = 0 to 8. Use that data to create a graph that plots speed-up vs. number of threads. This task must be completed on a machine with at least four processing cores. Discuss the scalability of your program.

You should submit a copy of your code as an ASCII text file and a PDF document with your results for Tasks 2.