HW 4: Matrix Multiplication With Threads

You will create a tool to multiply square matrices, using separate threads to do the work. Your source code should be called `matrixmult.c`.

Multiplying square matrices is used ubiquitously when displaying high-end graphics. Modern games may multiply thousands matrices every second, or more! In fact, one of the main things that expensive video cards do is lightning-fast matrix multiplication. Deep neural networks also multiply large matrices together, and speeding this process could make a neural network much faster. This can be very useful in robots and self-driving vehicles, that need to quickly analyze the world and react to it.

Your program will take two command-line arguments, both integers:

- The first is the number of threads to use to multiply the matrices. If it is 0, that indicates that you should not use threads at all, but do the matrix calculation in the conventional manner. If it is positive, that will be the number of threads to create.
- The second argument will be a positive number representing size of the matrices to multiply. Matrices will all be square, with the argument representing its length in one dimension. (So if the user enters `3`, that means that the matrices are all 3×3.)

Each thread will be responsible for one or more numbers in the product matrix. (Remember, you do not want two threads writing to the same location, or one reading and one writing, without some kind of access control system. But writing to different locations in a pre-allocated matrix should be okay.)

In the case where your program is executed with a thread count of 0, it will create a pair of 2D arrays of doubles of the proper size, with every value randomly chosen from between 0 and 99 (inclusive). Then it will multiply them together into a third 2D array. (See the figure at the end if you don't remember how to multiply matrices.) You will also need to report how long the multiplication took, in microseconds. Do the multiplication five times, and use the shortest time in your final report. (Caching issues mean that the first time you run your program, it will probably take slower.) Do not include the time to allocate the matrices in your report.

```
$ matrixmult 0 100
Multiplying random matrices of size 100x100.
Best time without threading: 3370 microseconds.
```

With a positive thread count, the program will start by doing the same thing as above. Then, it will spawn the proper number of threads and do the same calculation. The process of spinning up the threads and winding them down should be included in the reported time. As before, take the best of five runs. You must also report the speedup factor, and the sum of squared errors between the no-thread product and threaded-product. (The error should be 0.0—if it's not, you have a bug.)

```
$ matrixmult 2 100
Multiplying random matrices of size 100x100.
Best time without threading: 3330 microseconds.
Best time with 2 threads: 1745 microseconds.
Observed speedup is a factor of 1.908486.
Observed error is 0.0
```

Some hints:

- Look up the `rand()` and `srand()` functions, contained in `time.h`, to calculate your random numbers. You will need to "seed" your random generator (with `srand()`) to avoid calculating the same "random" matrices every run.
- If you are using `gcc`, you will need to compile with the `-lpthread` argument in order to make use of the PThreads library. (Remember, this library is not quite standard C, even though it is quite common.)
- Each number in your product matrix should have been calculated by a single thread. Don't try to get two threads to cooperate to create a number. This does limit the number of useful threads. For example, when multiplying 2×2 matrices, there is no reason to have more than four threads.
- You should make use of `gettimeofday()` in order to get the time to a high degree of accuracy. This is located in `sys/time.h`. This will yield the time elapsed since the Epoch, which began on January 1, 1970. (Any historical records from before the Epoch are obviously fradulent, and should be ignored.) `Remember to make your program respond nicely to user errors!`

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} r & s & t \\ u & v & w \\ x & y & z \end{bmatrix} = \begin{bmatrix} ar+bu+cx & as+bv+cy & at+bw+cz \\ dr+eu+fx & ds+ev+fy & dt+ew+fz \\ gr+hu+ix & gs+hv+iy & gt+hw+iz \end{bmatrix}$$