Homework assignment 2

Compile

Use `make all` to compile the 3 source files into binaries:

1. gauss_openmp
2. gauss_pthread
3. gauss_serial

Binary Usage

```
binary [N] [random seed] [# threads]
- N: Dimension of the matricies
- random seed: seed for populating the matricies
- # threads: number of threads to use (ignored for serial version)
```

Run Tests

To get a rough idea of performance differences you can use the built-in tests via `make test`. Otherwise you can use the binaries as described above.

Optimizations

- I tried normalizing the diagonals to 1 before performing the elimination, such as to reduce the total amount of divisions (an expensive operation), but the performance gain was not significant, and would have been a hardware dependent optimization anyway
- The provided serial algorithm performs the elimination operation on the entire row despite elements not in the upper triangular matrix being known to result in zeros or being used in the back-substitution phase. Instead I simply left those values to what they were initialized so that we don't have to waste resources on them.
- Some other minor changes to the program (some aren't C89 complaint but I'm sure everyone's using C11 by now)

Algorithm description

- The algorithm is similar to the provided serial algorithm
- for each diagonal element - e
  - use threads to apply elementary row operations on the rows below it that would eliminate the values in the column below e
  - for each row r, below e's row in it's own thread
    - apply subtract a multiple of e's row from r such that it would eliminate the element in the column below e in r
    - don't change the elements in or left of e's column
  - synchronize all threads

Correctness Argument

- Because of the barrier no two locations in memory are ever being written to simultaneously
- The only dependencies are that the rows & columns before the diagonal have already been processed, this is also handled by the barrier

Room for improvement

- The pthreads version currently assigns roughly equal work to all the processors via striping. This aproach is ok, however because the top of the matrix involves more work than the bottom, it might be better to assign rows in order to the processors. Some other approaches:
  - **Use a queue** - replace local `norm` with a statically defined `atomic_int` that each thread increments each step would be ideal and there is less time spent idling
  - **Assign consecutive elements to each thread** - will likely give same result as currently with potentially better cache efficiency with multi-socket machines
- Use SIMD vector instructions
- Blocking or other measures to improve cache efficiency (untested, possibly non-issue)
- It may be possible to combine the back-substitution phase with the elimination phase if we used pivots (which assignment mentioned we shouldn't use)
- I found that using -O3 significantly improves performance (proportionally for all algorithms)
- Probably a few more, but I have to write some essays and study for my midterms this week!