Name: Stephen Allegri

GTID: 903207717

Development/Structure Explanation: This program was relatively straight forward, with the majority of the complexity dealing with OpenMP and it's associated library. The program contains no helper file (this was for my own sake- trying to make it work with coc-ice), but there are still 5 independent functions outside of main() that perform specific tasks (like printing data, generating random numbers, etc.). In addition to montecarlo.c, I have created a CMakeList.txt file. I implemented the parallel computing as a round-robin system, where each thread generated iterations/nthreads points and compared them to the solved integrand value. This was a relatively simple implementation with OpenMP, and the parallel computing was approximately 2-4 times faster than serial for the ranges of N0 used.

```
double generate_random(int range);

double e_function(double x_rand);

void print_data(double thread_count, double N, double approx, double exact, double time);

double approximate_e_serial(int upper_y, int upper_x, int scale);

double approximate_e_parallel(int upper_y, int upper_x, int scale);
```

Testing procedure: Though it was pretty obvious that the parallel computing was running faster, I would perform some concrete tests to show differences between computing time. Utilizing omp_get_wtime() to keep track of computational time, I was able to track and compare the different run-times and prove that parallel computing was faster. More archaically, I would sometimes time the program myself with a stopwatch app on my phone for larger N0 values to see which finished fastest. Surprisingly, I had no performance hit with rand() as was suggested by other students.

```
int main() {

    int y_higher = 6;
    int x_higher = 1;
    int N0 = 10;

    srand(time(NULL));

    approximate_e_parallel(y_higher, x_higher, N0);

    approximate_e_serial(y_higher, x_higher, N0);

    return 1;

}
```

```
[sallegri3@login-coc-ice-1 ~]$ qsub test.pbs
26975.sched-coc-ice.pace.gatech.edu
```

Summary and Interpretation of Results: My personal results were good- in all cases, where N0 = 1, N0 = 10, and N0 = 100, the approximation function that utilized parallel computing performed better. As N0 scaled up to larger and larger values (I even tested this at N0 = 1000, which took minutes), parallel computing performed relatively better and better. This disparity can be seen below (N0 = 1: serial = 2.64 * parallel, N0 = 100: serial = 3.85 * parallel). On a final note, the Monte Carlo method for integration seems like an effective tool, but perhaps too computationally intensive compared some possible alternatives (I know there are other common integration methods that don't use random number generation). This method is a bit too slow for many different applications, FEM being one that comes to mind.