Problem 1. Parallelize this code to normalize a vector

Proof.
- It was interesting. I wasn't able to get more than a 4x speedup from parallelize with only one node. It got better with more nodes, but very slowly. Probably because of the large overhead and large amount of waiting.
- Pasted below are my code, makefile, and submission file

Listing 1: Parallel Normalize vector

```
#include <iostream>
#include <cstdlib>
#include <cmath>

#include <omp.h>
#define _USE_MATH_DEFINES

// function to compute the 2-norm of a vector v of length n
double norm(double *v, int n){
    double norm = 0.;

    for(int i=0; i<n; i++)
        norm += v[i]*v[i];

    return sqrt(norm);
}

// initialise v to values between -10 and 10
void initialize(double *v, int n){
    for(int i=0; i<n; i++)
        v[i] = cos(double(i)) * 10.;
}


void normalize_vector(double *v, int n){
    double norm = 0.;

    // compute the norm of v
    for(int i=0; i<n; i++)
        norm += v[i]*v[i];
    norm = sqrt(norm);

    // normalize v
    for(int i=0; i<n; i++)
        v[i] /= norm;
}

void normalize_vector_omp(double *v, int n)
{
    double norm = 0.;

    // compute norm of v
    #pragma omp parallel for reduction(+:norm)
    for (int i=0; i<n; i++) {
        norm += v[i]*v[i];
    }

    norm = sqrt(norm);
    // normalize v
    #pragma omp parallel
    {
      #pragma omp for
      for(int i=0; i<n; i++)
          v[i] /= norm;
    }
}
int main( void ){
    const int N = 40000000;
    double *v = (double*)malloc(N*sizeof(double));
    bool validated = false;

    initialize(v, N);
    double time_serial = -omp_get_wtime();
    normalize_vector(v, N);
    time_serial += omp_get_wtime();

    // chck the answer
    std::cout << "serial error : " << fabs(norm(v,N) - 1.) << std::endl;

    int max_threads = omp_get_max_threads();
    initialize(v, N);
    double time_parallel = -omp_get_wtime();
    normalize_vector_omp(v, N);
    time_parallel += omp_get_wtime();

    // chck the answer
    std::cout << "parallel error : " << fabs(norm(v,N) - 1.) << std::endl;

    std::cout << max_threads  << " threads" << std::endl;
    std::cout << "serial   : " << time_serial << " seconds\t"
              << "parallel  : " << time_parallel << " seconds" << std::endl;
    std::cout << "speedup  : " << time_serial/time_parallel << std::endl;
    std::cout << "efficiency : " <<
        (time_serial/time_parallel)/double(max_threads) << std::endl;

    free(v);
    return 0;
}
```

Listing 2: Make File

```
####################################

all: dot.exec \
     pi.exec \
     norm.exec
####################################


## Compile dot_prod
dot.exec: dot_prod.cpp
       g++ dot_prod.cpp -fopenmp -o dot.exec

## compile serial_pi
pi.exec: serial_pi.cpp
       g++ serial_pi.cpp -fopenmp -o pi.exec

## compile serial_pi
norm.exec: normalize_vec.cpp
       g++ normalize_vec.cpp -fopenmp -o norm.exec

clean:
       rm -rf *.exec
```

Listing 3: Submit file

```
#!/bin/bash
# a sample job submission script to submit an OpenMP job to the sandyb
# partition on Midway1 please change the --partition option if you want to use
# another partition on Midway1

# set the job name to hello-openmp
#SBATCH --job-name=Norm_vec

# send output to hello-openmp.out
#SBATCH --output=norm-vec-openmp.out

# this job requests node
#SBATCH --ntasks=2


# and request 8 cpus per task for OpenMP threads
#SBATCH --cpus-per-task=8

# this job will run in the sandyb partition on Midway1
```

```
#SBATCH --partition=sandyb


# set OMP_NUM_THREADS to the number of --cpus-per-task we asked for
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

# Run the process with mpirun. Notice -n is not required. mpirun will
# automatically figure out how many processes to run from the slurm options
### openmp executable
./norm.exec
```

☐

Problem 2. Parallelize this dot product code

Proof.

- It took about .7 seconds (un-parallelize) and then even longer .86 seconds when run parallelized this makes sense because of the overhead.
- It took about .4 seconds parallelized with 8 threads
- with an array of 500,000,000 , it took 2 seconds parallelized with 8 threads and my run with 1 thread was killed :(
- with array of 10,000, 8 threads took 1.934e-4 seconds and 1 thread took 9.45628e-05 seconds. This is wild because fewer threads took less time. Amdahl's law strikes again!

Listing 4: Parallel Dot product vector

```
#include <iostream>
#include <vector>

#include <omp.h>

int main(void){
    const int N = 10000;
    std::vector<double> a(N);
    std::vector<double> b(N);

    int num_threads = omp_get_max_threads();
    std::cout << "dot of vectors with length " << N << " with " << num_threads <<
        " threads" << std::endl;

    // initialize the vectors
    for(int i=0; i<N; i++) {
        a[i] = 1./2.;
        b[i] = double(i+1);
    }

    double time = -omp_get_wtime();
    double dot=0.;
    #pragma omp parallel for reduction(+:dot)
    for(int i=0; i<N; i++) {
        dot += a[i] * b[i];
    }
    time += omp_get_wtime();

    // use formula for sum of arithmetic sequence: sum(1:n) = (n+1)*n/2
    double expected = double(N+1)*double(N)/4.;
    std::cout << "dot product " << dot
              << (dot==expected ? " which matches the expected value "
                                : " which does not match the expected value ")
              << expected << std::endl;
    std::cout << "that took " << time << " seconds" << std::endl;
    return 0;
}
```

□

Problem 3. MonteCarlo Estimate Pi

Proof.

Listing 5: Parallel Dot product vector

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

int main()
{
    const int niter = 10000000;
    double x,y;
    int count=0;
    double z;
    double pi;
    int threadID;

    //main loop
    #pragma omp parallel private(x, y, z, threadID)
    {
      threadID = omp_get_thread_num();
      srand(threadID);
    #pragma omp for reduction(+:count)
    for (int i=0; i<niter; ++i) {
        //get random points
        x = (double)random()/RAND_MAX;
        y = (double)random()/RAND_MAX;
        z = sqrt((x*x)+(y*y));
        //check to see if point is in unit circle
        if (z<=1)
        {
            ++count;
        }
    }
    /* end omp parallel */
    }
    pi = ((double)count/(double)niter)*4.0;      //p = 4(m/n)
    printf("Pi: %f\n", pi);
    return 0;
}
```

□