Exercises 1

Coauthored by: Teng, Chao && Jonas Karl, Böcker

Problem 1

The nonparallel version of computing pi by Monte Carlo.

```
void compute_pi_monte_carlo(int n_steps)
{
    int m = 0;
    unsigned int seed = time(NULL) ^ omp_get_thread_num();
    srand(seed);

    for (int i = 0; i < n_steps; i++)
    {
        double x = (double) rand() / RAND_MAX;
        double y = (double) rand() / RAND_MAX;
        if (x * x + y * y < 1.)
        {
            m += 1;
        }
    }

    double pi = 4.0 * m / n_steps;
    printf("Estimated value of pi = %f\n", pi);
}
```

Problem 2

Here, I implemented a OpenMP version of Monte Carlo. I used `reduction` to avoid **data race** for `m`.

```
void compute_pi_monte_carlo_parallel(int n_steps)
{
    int m = 0;
#pragma omp parallel
    {
        unsigned int seed = time(NULL) ^ omp_get_thread_num();

#pragma omp for reduction(+:m)
        for (int i = 0; i < n_steps; i++)
        {
            double x = (double) rand_r(&seed) / RAND_MAX;
            double y = (double) rand_r(&seed) / RAND_MAX;
            if (x * x + y * y < 1.)
            {
                m++;
            }
        }
    }

    double pi = 4.0 * m / n_steps;
    printf("Estimated value of pi = %f\n", pi);
}
```

Test Wall Time Elapsed for Our Programs

To test how good our parallel program performs, we need to measure the **Wall Time** (Time consumed in the real world) of our program instead of CPU Time. For the measurement of this, the recommended way is the Linux/MacOS command `time` or the function `omp_get_wtime` provided by `OpenMP` other than `clock`.

I use the following code in my code for the measurement purpose (use `clock` if you wish to compile without `OpenMP`):

```
int n_steps = 100000000;
double start = omp_get_wtime();
...Your Code/Function For Parallel Computation
double end = omp_get_wtime();
double time_spent = end - start;
printf("Wall time taken: %f seconds\n", time_spent);
```

The testing result when we set `n_steps=100000000` :

```
# Parallel version
Estimated value of pi = 3.141505
Wall time taken: 0.229149 seconds
-----------------------------------
# Nonparallel version
Estimated value of pi = 3.141704
Wall time taken: 1.495120 seconds
```

Problem 3

In this part we implement a MPI version of the Monte Carlo example.

First make sure we have installed OpenMPI properly:

```
mpicc --version # For compiling MPI program (We can also edit CMakeLists.txt to enable gcc to compile it)
mpiexec --version # For executing MPI program
```

Include `mpi.h` to make MPI functions visible to our code. In MPI program, `rank` is the id of the current thread that is doing the parallel tasks; `size` indicates how many threads are running in parallel. To boot up a MPI program, we use `MPI_Init(argc, argv)` at first.

Initialize MPI and get the parameters of our parallel program:

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Similarly, we assign a random seed for each thread and use `rand_r()` for random number generation. We collect statistics respectively for each thread,

```
    srand(rank);

    for (int i = 0; i < amount; ++i) {
        const double x = (double) rand() / RAND_MAX;
        const double y = (double) rand() / RAND_MAX;

        if (x * x + y * y <= 1)
            ++m;
    }
```

and reduce after all the jobs are done.

```
    MPI_Reduce(&m, &total_hits, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("PI is approximately equal to: %f\n", 4. * total_hits / n);
    }
```

Put `MPI_finalize()` in the end of the MPI program to free the threads.

To execute MPI program with 4 threads, we use:

```
mpiexec -np 4 your_program_fullname
```

The result of our program:

```
Process 3 out of 4
Process 0 out of 4
Process 2 out of 4
Process 1 out of 4
Estimated Pi = 3.1414960000
```

Problem 4

See the pages after the Problem 6.

Problem 5

My implementation of parallel scalar product between vector and vector:

```
    double a[5] = {1.0, 2.0, 3.0, 4.0, 5.0},
           b[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
    double res = .0;

#pragma omp parallel for reduction(+:res)
    for (int i = 0; i < 5; i++) {
        res += a[i] * b[i];
    }

    printf("Dot product = %f\n", res);

    return 0;
```

Suppose we use n processors for vector of order n, the time complexity of doing multiplication is $O(1)$ while the complexity of reduction is $O(\log n)$, when we *divide and conquer*. As a result,

$$T(n) = O(\log n).$$

Problem 6

My implementation of parallel dot product between matrix and vector:

```
    // dot product mat and vec
    double mat[2][2] = {{1.0, 2.0}, {3.0, 4.0}},
           vec[2] = {1.0, 2.0};
    double res2[2] = {0.0, 0.0};
    double start = omp_get_wtime();

    // collaspe(2) is used to collapse the two loops into one
#pragma omp parallel for collapse(2) reduction(+:res2[:2])
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            res2[i] += mat[i][j] * vec[j];
        }
    }
    double end = omp_get_wtime();
    double time_spent = end - start;
    printf("Wall time taken: %f seconds\n", time_spent);
    printf("Matrix-vector product = [%f, %f]\n", res2[0], res2[1]);
```

Each row of the resulting vector is calculated in the same way with vector dot product, thus having a time complexity of $O(\log n)$. In the reduction phase, we still can reduce the results in parallel, which results in a $O(\log n)$ time complexity. What is different is that we need $n \cdot n$ processors for the reduction phase. Hence,

$$T(n) = O(\log n).$$

But....

This seems true. However, in EREW model, exclusive read is not allowed so we can not do n vector inner product at the same time so the time complexity should still be $O(n \log n)$.