Report: Homework 2 hello {omp, mpi}

Task 1

- OpenMP 4.0 supporting compiler installed: gcc (GCC) 4.8.2, and g++ (GCC) 4.8.2
- MPI implementation installed: MPICH2 1.0
- Operating system: macOS High Sierra
- Processor speed: 2.4 GHz
  Number of processors: 1
  Total number of cores: 2
  L2 Cache (per Core): 256 KB
  L3 Cache: 3 MB
  Memory 4 GB

Task 2

1. Perform the variable declaration, **int size, rank;** before the pragma:
   - The code does not work as the original: the **rank** are incorrect.
   - Explain: this is because the variable **rank** is a shared variable which is accessed by all threads. After a thread assigns its value for **rank**, this value may not be printed out yet before another thread comes in to change it. That is why the displayed values of **rank** are not as the original code. This also happens for **size**. However, we could not see the behavior as in the case of **rank** because **size** is always 8 (e.g. if we use 8 threads to run the code).

2. Compute **size** before the pragma:
   - The code always prints out 1 for **size** instead of total number of threads used to run the code.
   - This is because the function **omp_get_num_threads()** returns the number of threads that are currently doing something. When it is called in a serial region (i.e. before **pragma**) it will return 1 because only 1 process is working in serial region. If it is called in a parallel region it will return the number of threads that are being used as we saw in the original code.

3. Maximum number of threads that I can run on my machine is 2048. I actually do not understand why a mac gives this limit. I got this number by trying to increase the number of threads up to the point when I cannot go higher anymore.

For a machine with Linux OS, we can find this limit by

```
cat /proc/sys/kernel/threads-max
```

4. The code for this question is in the folder **/hw2_omp** which is attached to this homework submission. Please refer to README for instructions how to build and run the code.

The running time with different number of threads is shown in the Table below.

Number of threads	1	2	4	8	16	32	64	128
Time (s)	1.107305	0.943376	0.772639	0.764670	0.763840	0.763476	0.798328	0.759413

Comments:

- The running time is faster when there are more threads to be used.
- However, this speed-up is not as fast as we should have. This is because of the using rand() function to generate random number. The rand() function is shared with all threads. To test this argument, I replace the A[i]=rand()%100 by A[i]=1 (i.e. just assign a fixed integer for all members of the array). This changing results the following running time. As seen, the speed-up when increasing number of threads is much faster than before.

Number of threads	1	2	4	8	16	32	64	128
Time (s)	0.488123	0.284846	0.214155	0.186146	0.176061	0.169647	0.167763	0.169930

- To improve this, we could implement a way to generate random number locally in thread, i.e. each thread has its own generator. We could get more information about this at https://stackoverflow.com/questions/37305104/how-to-generate-random-numbers-in-a-thread-safe-way-with-openmp
- However, I think this homework just aims to the introduction to OpenMP. Therefore I will go further with this random generator at a later time.

Task 3

The hybrid MPI+OpenMP code is in the folder /hw2_mpi which is attached to this homework submission. Please refer to README for instructions how to build and run the code.