Homework 2

- Yucheng Zhang

0. Machine

- MacBook Pro 2017
- Compiler version: `g++ (Homebrew GCC 8.2.0) 8.2.0`.
- CPU: `Intel(R) Core(TM) i7-7700HQ CPU @ 2.80Ghz`.
  - Details: https://ark.intel.com/products/97185/Intel-Core-i7-7700HQ-Processor-6M-Cache-up-to-3-80-GHz-.
  - Cores: `4`, Threads: `8`.
  - Max turbo frequency: `3.80 GHz`.
  - Operations per cycle: `16 DP FLOPs/cycle` for Intel Kaby Lake as found here https://stackoverflow.com/a/15657772.
  - Max flop rate: `Cores * Max turbo frequency * Operations per cycle = 243.2 Gflop/s`.
  - Max memory bandwidth: `37.5 GB/s` for 2 channels.

1. Finding Memory bugs.

1. Index exceeds the array range; `free()` should be used for memory allocated by `maclloc`.
2. Variable is used before initialized, while valgrind does not complain about this.

2. Optimizing matrix-matrix multiplication.

- Try different loop arrangements.
  - The outputs are shown in files/2-order-jpi.txt, files/2-order-jip.txt and files/2-order-ipj.txt.
  - We can see that the performance of order jpi is the best. This is because the matrices are stored in column major order. For order jpi, `double A_ip = a[i + p * m]; double B_pj = b[p + j * k]; double C_ij = c[i + j * m];` are all read in continuous memory location, which can save a lot of time.
  - Similarly, we get the worst performance for order ipj, which would be the best if the matrices were stored in row major order.
- Implement a one level blocking scheme by using BLOCK_SIZE macro as the block size.
- Experiment with different values for `BLOCK_SIZE`.
  - From the table below, we can see that we get better performance around `BLOCK_SIZE = 64`.

`BLOCK_SIZE`	`Gflop/s` (Average)	`GB/s` (Average)
`4`	`6.089491`	`97.431855`
`8`	`3.242940`	`51.887047`
`16`	`2.426508`	`38.824121`
`32`	`15.953180`	`255.250880`
`64`	`19.541914`	`312.670619`
`128`	`16.749192`	`267.987077`
`256`	`14.572623`	`233.161976`

- Parallelize your matrix-matrix multiplication code using OpenMP.
    - I parallelize the code on the for loop over blocks in C.
    - One thing to notice is how the cache is shared by all the threads. The optimal `BLOCK_SIZE` may be different when we use different number of threads. For example, when I use OpenMP with more than one thread, `BLOCK_SIZE = 32` works better than `BLOCK_SIZE = 64`, which is optimal in the serial case.
- What percentage of the peak FLOP-rate do you achieve with your code?
    - I can achieve up to `22.6 %` of the peak FLOP-rate.

3. Finding OpenMP bugs.

2. `reduction` should be used for simple sum. `int` and `float` may not be large enough in some case, which might depend on the machine.
3. For a function which may not be excuted by all threads, `#pragma omp barrier` inside may cause the program to get stuck.
4. `private` stack size is not very large.
5. `lock` may cause the program to get stuck if not used properly.
6. We may use global variables in order to be shared easily.

4. OpenMP version of 2D Jacobi/Gauss-Seidel smoothing.

- The following tables show the timings for different values of `N` and different numbers of threads. Number of iterations is `100`.
- Jacobi method

`N_thread`	`N=100`	`N=1000`	`N=10,000`	`N=20,000`
`1`	`0.002660 s`	`0.209642 s`	`19.760169 s`	`81.583017 s`
`2`	`0.004587 s`	`0.163366 s`	`13.533498 s`	`54.148035 s`
`4`	`0.005771 s`	`0.136099 s`	`12.947494 s`	`53.681173 s`
`8`	`0.007180 s`	`0.143475 s`	`13.939527 s`	`69.407606 s`

- Gauss-Seidel method

`N_thread`	`N=100`	`N=1000`	`N=10,000`	`N=20,000`
`1`	`0.001833 s`	`0.204328 s`	`23.346515 s`	`94.526992 s`
`2`	`0.006006 s`	`0.186694 s`	`19.875444 s`	`78.298179 s`
`4`	`0.009349 s`	`0.177580 s`	`19.546787 s`	`79.597596 s`
`8`	`0.011964 s`	`0.199010 s`	`20.717127 s`	`81.617238 s`

- We can see that for large `N`, `2 threads` work better than `1 thread`, but more threads (`4` and `8`) don't really perform better. I think it's related to the memory bandwidth and also the fact that I'm using other softwares during the timing.
- I also test the code on anther machine, which has `Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz`, with `24` cores and `1` thread per core. Timings are summarized below, with `100` iterations and `N=20,000`.
- Jacobi method

`N_thread`	`1`	`2`	`4`	`8`	`16`	`24`
`Time (s)`	`137.929256`	`66.865910`	`38.367468`	`25.794236`	`21.755469`	`21.025227`

- Gauss-Seidel method

`N_thread`	`1`	`2`	`4`	`8`	`16`	`24`
`Time (s)`	`172.466722`	`81.096346 s`	`43.46177`	`26.440732`	`21.187021`	`21.504986`

- We see that on this more stable machine, we can reduce the time to half if we double the number of threads until `N_thread = 8`. For more threads, the performance may be restricted by the cache size and bandwidth.