Homework 3
Han Tran

Task 1

With **#pragma omp parallel for**, the compiler will distribute the for-loop equally to the threads. For example, running with 4 threads, then thread 0 handles indices i=0-3, thread 1 handles indices i=4-7, … If the size of for loop is not a multiple of the number of threads, then the remaining (after modulus division) is distributed to the threads 0 up to the thread right before the last one.

With the **for** removed, then every thread will do the same thing, i.e. each will have the loop from the first index to the last index.

Parallel for might be useful when the for loop handle independent tasks (i.e. the tasks in next loop do not depend on the tasks in previous loop) and we do not want to spend much effort to interfere the work distribution of the for loop. The distribution is hand-overfed to the compiler.

However, we might want to use **parallel** without for even we do have a for loop in the parallel region. This is for the cases when we really want to control the distribution of work to the threads in a designated way to obtain a better performance.

Task 2
(# threads used = 16)

	Running time (s)
Original code	25.590716s
1. Not specifying any scheduling	39.798989s
2. Using static scheduling for the 2nd loop	25.755565s
3. Modifying the chunksize	Time does not change

Explanation:

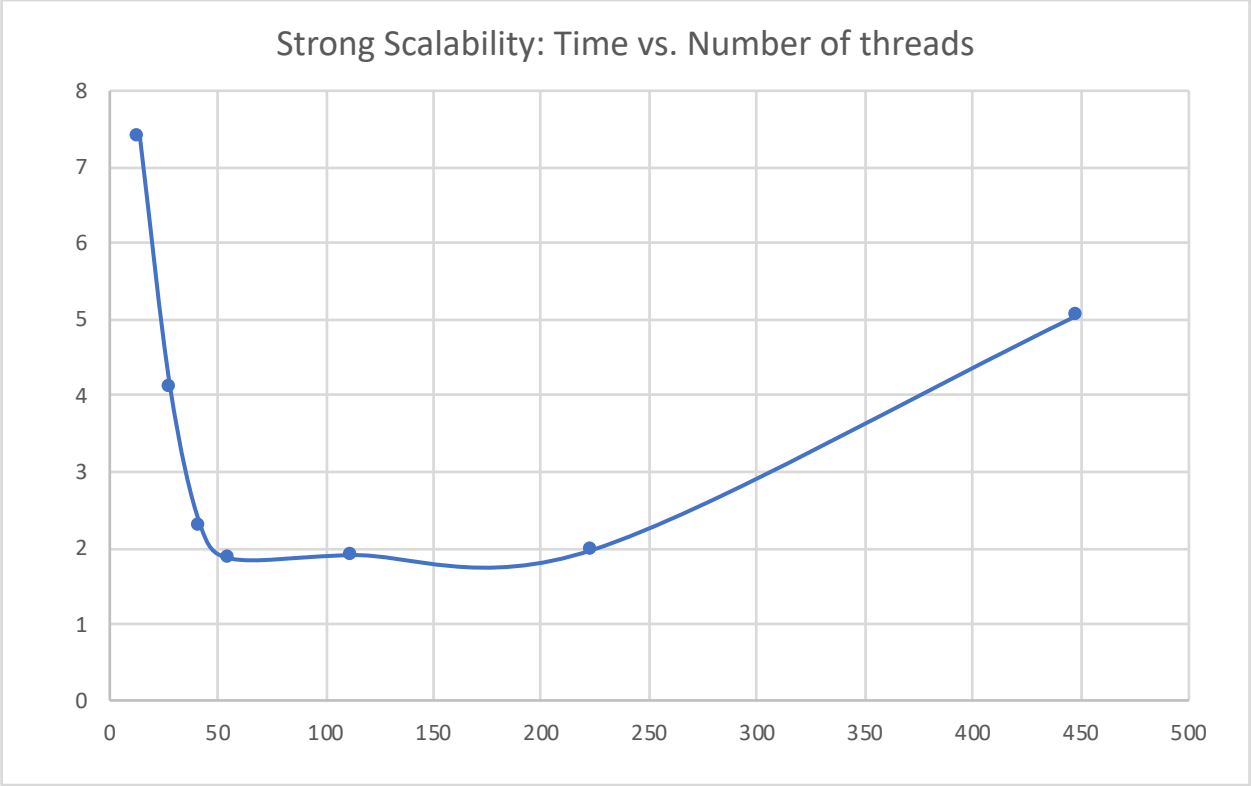
- As seen, when no scheduling is used then the code runs slower. This slow comes mostly from the 2nd loop. When no scheduling, the iterations are continuously distributed to threads, i.e. thread 0 will handle indices 0, 1, 2, …k; thread 1 will take indices k+1, k+2, … Thus, it is obvious that for the 2nd loop, thread 0 will take the least workload because it just computes the small powers. The workload increases with thread 1, 2, … This means that thread 0 finishes its job quickly and stay there to wait for higher threads do their job. That is why if we use dynamics scheduling, once thread 0 finishes, it will be dynamically assigned another job.
- If using static scheduling for the 2nd loop, then the time is back to similar to the original code. This is because in static scheduling, the iterations are distributed in round-robin fashion, i.e. indices 0, 1, 2, 3, … are assigned for threads 0, 1, 2, 3, … respectively, up to the index chunksize, then it starts a new round. Therefore, even it is static, but the

workload for every thread is nearly equal in the 2nd loop. That is why there is no difference between unsing static and dynamic scheduling for the 2nd loop.

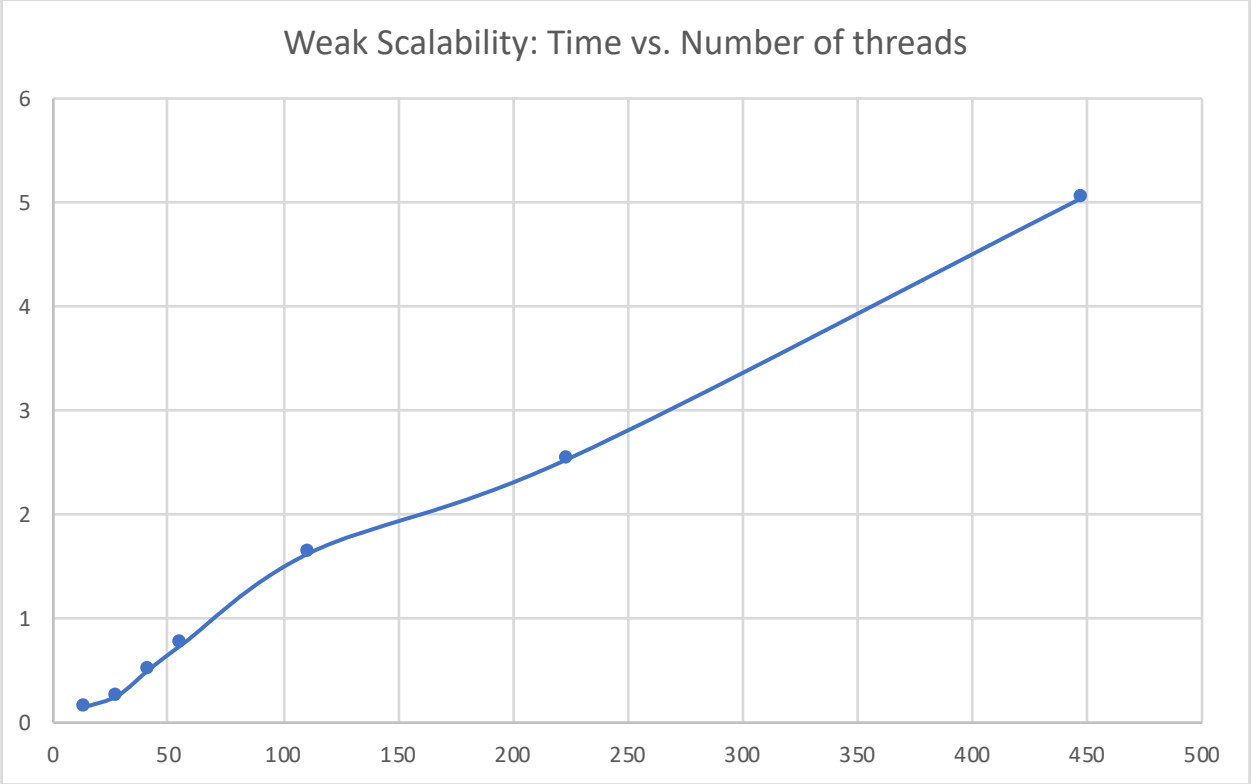
- The running time does not change when the chunksize is modified. This is because the number of threads is smaller than the number of iterations in for loop. Thus, all threads are used with all of their power whatever the chunksize is.

Task 3

Strong scalability experiment is shown in the following figure. For this experiment, I use n = 80,000 and m = 50,000. I run the code with number of threads np = 14, 28, …,448 as shown on the x-axis of the chart and measure the running for the parallel matrix-vector multiplication. The running time at first deceases when np increases as we expected. However, when np getting higher, the time starts increasing. This is due to the fact of getting complex communication when the number of threads is too large.



Weak scalability experiment is shown in the following figure. For this experiment, the ratio (n*m/np)=(4E9/448) and to be kept as constant. I run with different number of threads as shown on the x-axis of the chart. It can be seen that the time increases when number of threads increases. Ideally, we should have the time to be unchanged. However, in reality, when np increases, the communication also become more and more complicated. That will make the code run slower even the ratio (size/np) does not change. This is what we can see in this experiment.



Task 4

The code is attached with this homework submission.

- For the storage of sparse matrix, I use Compressed Row Storage (CRS) format: http://netlib.org/linalg/html_templates/node91.html#SECTION00931100000000000000
- For the sparse matrix, I download the matrix 1138_bus from https://sparse.tamu.edu. The matrix is save in text file **1138_bus.txt**. This file will be read by the program to load the data of the matrix.
- To test the code: in the main program, I generated an identity matrix **I** (in CRS format) and call the SpMv function to do the multiplication **y** = **Ix**. Then to check if **y**-**x** < tol where tol is the given tolerance (=10^{-4}). If all are less than the tolerance, the test is passed.

Task 6

The MPI code is attached with this homework submission.

The two versions of using blocking and non-blocking of send and recv are written in the same code. Currently the code is using non-blocking version. To use blocking version, we need to comment out {MPI_Isend, MPI_Irecv, MPI_Wait} and un-comment {MPI_Send, MPI_Recv} in the main program.