ECE 5720 HW2

Chen-Tung Chu cc2396

1. General Remarks

In this assignment, our goal is to find the inverse matrix of a matrix A, with two different parallel approaches: *pthread* and *openmp*.
Our method in finding such inverse matrix is using Gaussian elimination with partial pivoting.
In short, given a matrix A, b, such that $Ax = b$, we want to solve x. By implementing the Gaussian elimination, we cover two parts of the solving process:

- Make the form of triangularization:
  - First, we swap the two rows.
  - Then we multiply row by a nonzero number.
  - Last, we add a multiple of one row to another row.
- Solve the upper-triangle systems by using backsubstitution

After the solving process, now we get the inverse matrix of A (A^{-1}) .
The first parallel method on solving such problem is using *pthread*.

2. Using *pthread*

In my code (cc2396_hw2_pthread.c), In the main function, with the thread declarations and the initial setup of barrier, we can partition the data and work it in synchronization.

```
      pthread_t thread[num_thrs];
      pthread_barrier_init(&barrier, NULL, num_thrs);
```

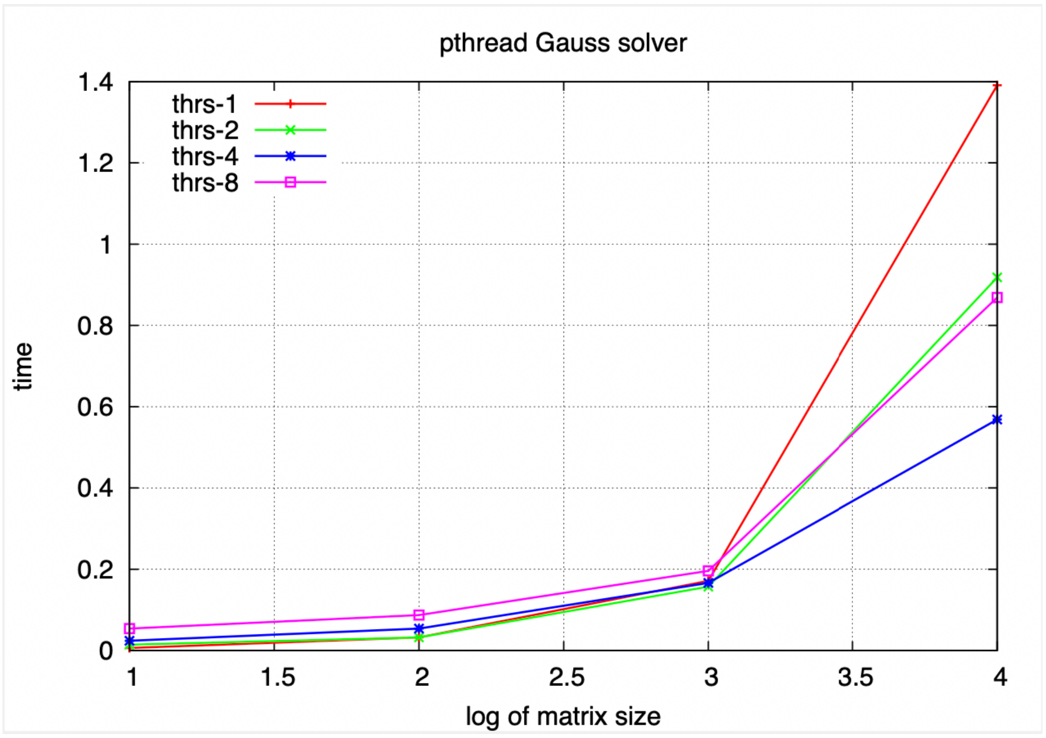
After populating A and b matrices, we now have b as the identity matrix. Now we can compute the inverse matrix with activating the threads, again, the first thing we need to do is triangularization. After the triangularization process, we will proceed to backsubstitution. During the triangularization process, we need to ensure that the triangularization process must be finished, then the backsubstitution can join. So we have to terminate the thread respectively.

```
      // activate threads for triangularization of A and update of b
      for (ii = 0; ii < num_thrs; ii++) {
        pthread_create(&thread[ii], NULL, triangularize, (void *) 
&index[ii]);
      }

      // terminate threads
      for (ii = 0; ii < num_thrs; ii++) {
        pthread_join(thread[ii], &status);
      }

      pthread_barrier_destroy(&barrier);

      // backsubstitution, A is now upper triangular, b has changed too
      //gettimeofday(&start, NULL);

      // activate threads for backsubstitution
      for (ii = 0; ii < num_thrs; ii++) {
        pthread_create(&thread[ii], NULL, backSolve, (void *) &index[ii]);
      }
      // terminate threads
      for (ii = 0; ii < num_thrs; ii++) {
        pthread_join(thread[ii], &status);
      }
```

Once we finish the computation, we can now compute the residual error to track the accuracy of paralleling. First I defined the minimum and maximum dimension of the matrices as follows:

```
# define MIN_DIM     1<<7         // min dimension of the matrices (equal to
MIN_DIM if Nrhs=1)
# define MAX_DIM     1<<10        // max dimension (equal to MAX_DIM if
```

With the first approach, I have the following result of the calculation time chart (gauss_plots_pthread.eps) and table (Gauss_solver_pthread.csv).



From the chart we can observe that the with the same matrix size, the workload seemed not very distributed since when we use eight threads comparing to four threads, the calculation time did not become the smallest, it can be understood by the fact when the triangularization process, when the cyclic distribution of rows among threads, as *i* is close to *n*-1, it would become insufficient since almost all the work has been done, In comparison, we would rather use a single thread at this particular time, so with the chart, the best average performance would be using four threads.

The following table is the total compute time of using different threads:

Number of threads	Mat_size = 1	Mat_size = 2	Mat_size = 4	Mat_size = 8
1	7.342e-03	9.68e-03	7.956e-03	1.6e-02
2	2.872e-02	3.675e-02	4.949e-02	9.5e-02
4	1.827e-01	1.67e-01	1.799e-01	2.052e-01

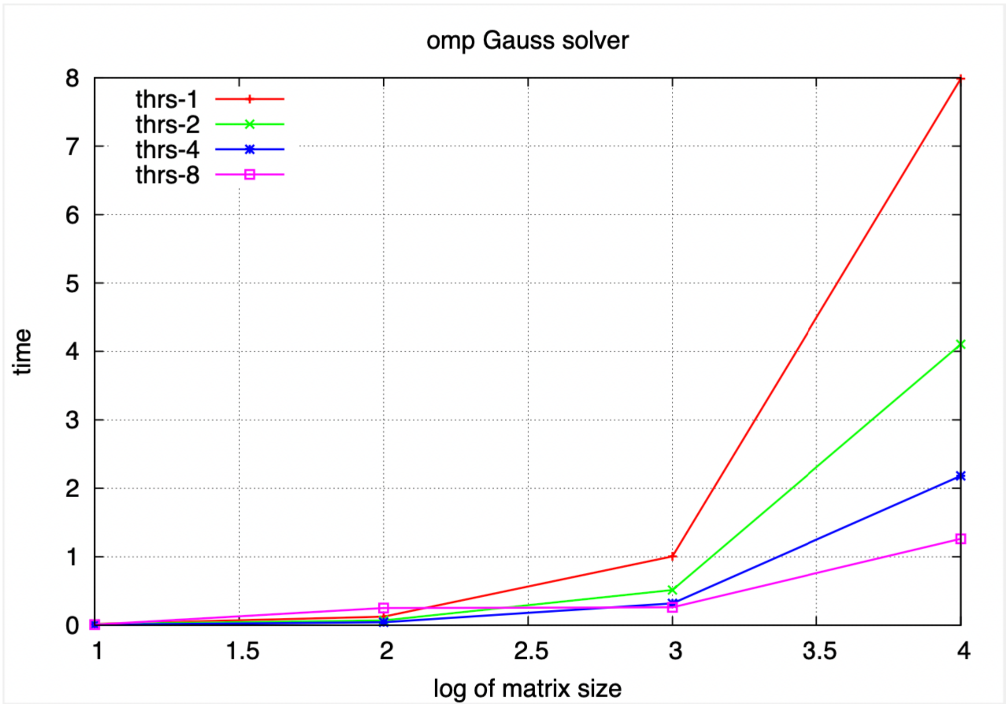
8	1.428e+00	9.384e-01	6.538e-01	1.058e+00

3. Using *openmp*

In cc2396_hw2_openmp.c, to make an easy comparison, I set the minimum and maximum dimension of the matrices the same as I used in *pthread*.
The main function in *openmp* is almost similar to *pthread*, the initialization is basically the same since we are solving the same problem. The difference is the setup of the threads. After initializations like allocating memories for matrices, in the main function, I implemented the original ways of solving the inverse matrix with triangularization and backsubstitution. With these processes, I recorded the total process time. The following is a piece of code presenting the inversion process of the matrix in main functuion.

```
    gettimeofday(&start, NULL);

    printf("Start Iversion\n");
    triangularization_omp(N,A,b,nrhs);
    backsubstitution_omp(N,A,b,x,nrhs);
    printf("Finished\n");

    gettimeofday(&end, NULL);

    el_time = ((end.tv_sec  - start.tv_sec)*1000000u +
                end.tv_usec - start.tv_usec)/1.e6;

    printf("total time is %10.3e seconds\n",el_time);
    fprintf(fp, "%1.3e ", el_time);
```

Both triangularization and backsubstitution methods I used are basically the same as I implemented in *pthread*. One of the advantages of using the same method is that I can get a better accuracy and total execution time in comparison.
For the experiment result, the final chart and table are as follows:

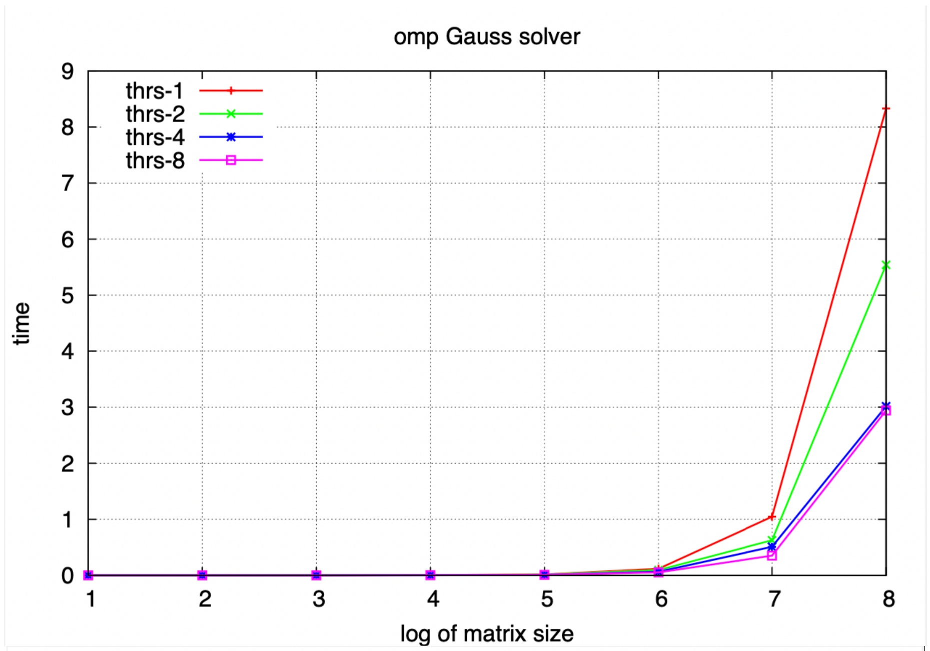


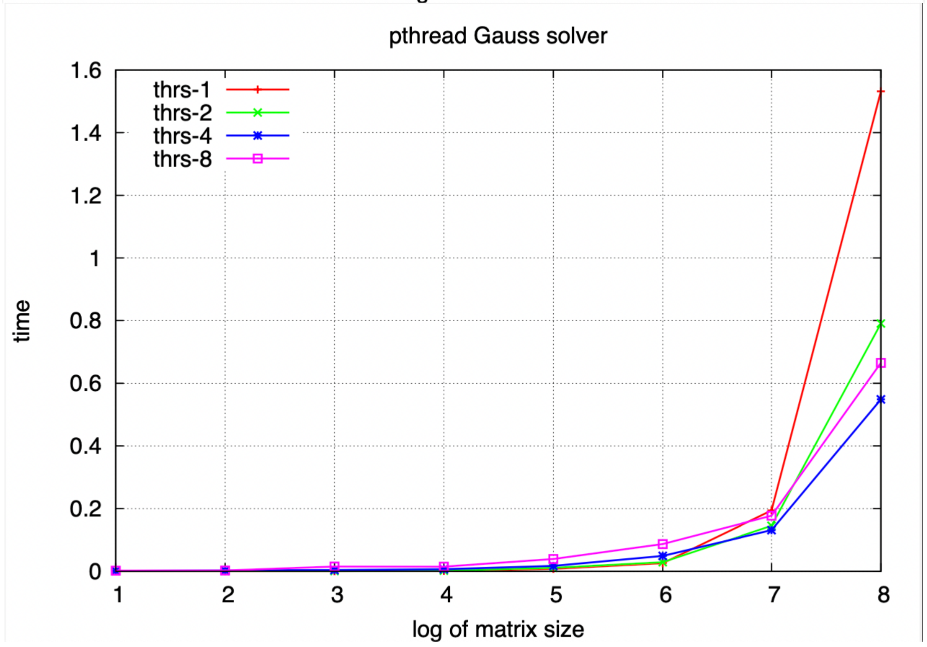
With *openmp,* we can observe that the average process time for single thread has significantly larger than using multiple threads when the matrix size increases. And the following table is the process time using different threads.

Number of threads	Mat_size = 1	Mat_size = 2	Mat_size = 4	Mat_size = 8
1	1.824e-02	1.766e-02	5.802e-03	8.527e-03
2	1.234e-01	6.856e-02	4.279e-02	2.5e-01
4	1.005e+00	5.133e-01	3.177e-01	2.602e-01
8	7.986e+00	4.106e+00	2.183e+00	1.262e+00

4. ***Openmp vs. pthread***

I think the major difference of implementing *openmp* and *pthread* is the convenience of using the API. For a simple comparison, when we set up threads, *openmp* simply needs to set up omp_set_num_thread(num_thrs) then it can acces its parallelism in the function by simply call # pragama omp parallel. This is much easier in programming than *pthread.* Moreover, when using *openmp*, we can observe that in *openmp* implementation, it is beneficial to use multiple threads. While in *pthread*, as observed and metioned earlier, we can find out sometimes it is not beneficial when *i* is close to *n-1*. Figures below can show the comparison when the log matrix size from three to ten, *openmp* distributed the workload quite evenly.





From the figures above, we can see that with same matrix size, *pthread* does not distributed well compared to *openmp* since the performance of eight threads was not better than four threads.

However, it seems that *openmp* does not perform well than *pthread.* The reason I reckon might because *openmp* want to distribute the threads more evenly, so it might cost more time to operate. Below is the table of operation time comparison (only several operations comparison).

pthread	1.45e-01	1.32e-01	1.77e-01	1.53e+00	7.91e-01	5.48e-01	6.65e-01
openmp	5.50e-01	2.81e-01	1.95e-01	7.95e+00	4.13e+00	2.36e+00	1.32e+00

The other reason may be that when using *openmp*, *n* threads running n loops in its entirety. So when *n* threads running *n* loops at the same time as one single thread running one loop.

5. Reference

[1] https://en.wikipedia.org/wiki/Gaussian_elimination
[2] https://people.cs.rutgers.edu/~venugopa/parallel_summer2012/ge.html
[3] https://quick-adviser.com/why-is-openmp-slower/
[4] Course lecture