CSCI551
German Razo

Gaussian Elimination with OpenMP

➢ Data Storage

- **Augmented matrix**: I allocated the augmented matrix **A | b** in a double pointer array variable called matrix of size n, rows of the augmented matrix were allocated with size n + 1, so A and b are combined in an array. So, if n was 4, then rows would have size 5. I also allocated a temporary matrix stored in a variable called **c_m** with the same size as the original augmented matrix as mentioned before, so **c_m** is a copy of the augmented matrix**.**
- **Result of unknowns:** to store the values of the unknowns that were calculated from the Gaussian elimination I allocated a one-dimensional array of type double.
- **L2-norm:** To calculate the result of L2norm I used the original augmented matrix stored in the variable called **matrix** and the allocated array with the values of the unknowns that were calculated with the Gaussian elimination.

➢ Data partition and work and exploit parallelism

- Data partition and parallelization work was used in two sections of my code; first section is in the function called **forward elimination** and second section is in the function called **back substitution**.

    - **Forward elimination**: Inside this function I solved for forward elimination, and because forward elimination requires the product of two rows to implement the resultant upper triangular I used two for loops to solve this step. However, I analyzed a way I could exploit parallelism here because is two for loops that will iterate (n) * (n + 1) times, and it is a good section to exploit parallelism, so I added the directive **# omp parallel for** before the **two for loops** to divide every row among each **thread** so no execution gets idle. So, every thread contributes to perform the product of two rows to produce the resultant upper triangular matrix and there is a balance of the amount of work each thread should perform.

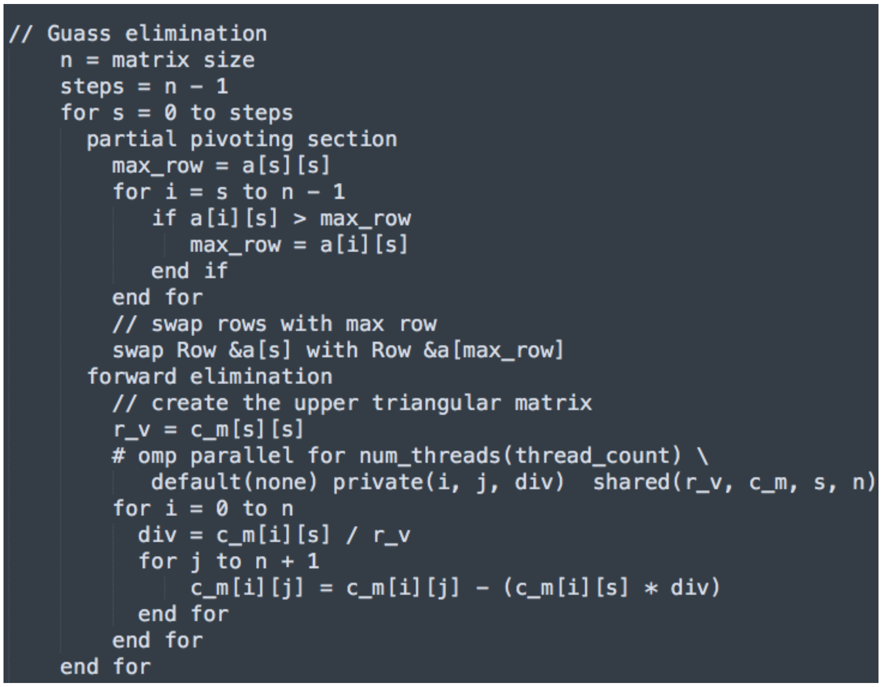
    - **Back substitution:** Inside this function I solved for back substitution, and I realized this is another section in my code to exploit parallelism because it iterates n times the number of rows and it requires two for loops. However, to obtain better performance I used column oriented instead of row oriented so every **thread** calculates the product of every column. In back substitution, the inner loop will iterate the most compare to the outer loop, so for this reason I added the directive **# omp parallel for** before the inner for loop. This way, inner loop exploits parallelism and gives better performance than row oriented.

How the parallel work is synchronized

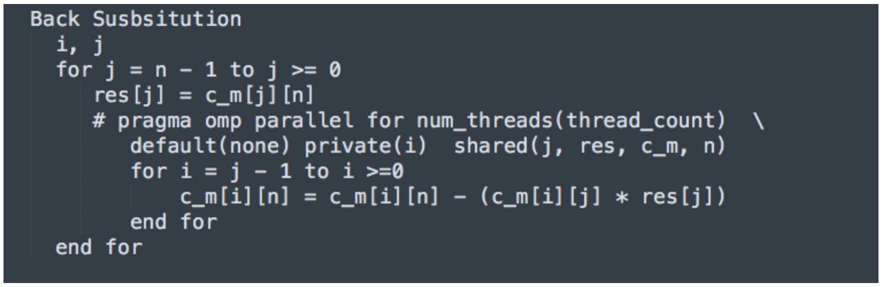
- When the program starts, only a single thread (**master thread**) is running, however, when we add directives like # omp parallel and clauses like **num_threads** and when our program reaches that part of the code our program then executes with num_threads – 1 the code inside the # omp parallel directive. After each thread, has completed running that section of the code, waits for all threads to finish because there is an **implicit barrier** that ensures that all threads wait for other threads to complete that section of code.

- When the directive # pragma omp parallel for is added, Synchronizations happens at the end of the parallel block where all the threads join after the directives # pragma omp parallel. After all threads in the block of code inside # pragma omp parallel have finished, the master thread continues executing until the end of the program.

Pseudocode of key elements of the parallelization strategy



```
// Guass elimination
    n = matrix size
    steps = n - 1
    for s = 0 to steps
      partial pivoting section
        max_row = a[s][s]
        for i = s to n - 1
           if a[i][s] > max_row
              max_row = a[i][s]
           end if
        end for
        // swap rows with max row
        swap Row &a[s] with Row &a[max_row]
      forward elimination
        // create the upper triangular matrix
        r_v = c_m[s][s]
        # omp parallel for num_threads(thread_count) \
           default(none) private(i, j, div)  shared(r_v, c_m, s, n)
        for i = 0 to n
          div = c_m[i][s] / r_v
          for j to n + 1
              c_m[i][j] = c_m[i][j] - (c_m[i][s] * div)
          end for
        end for
    end for
```



```
Back Susbsitution
  i, j
  for j = n - 1 to j >= 0
     res[j] = c_m[j][n]
     # pragma omp parallel for num_threads(thread_count)  \
        default(none) private(i)  shared(j, res, c_m, n)
        for i = j - 1 to i >=0
            c_m[i][n] = c_m[i][n] - (c_m[i][j] * res[j])
        end for
  end for
```

➢ what compiler/linker flags you chose to use and why

icc -Wall -W -Werror -qopenmp -O3   -std=c99 –o gauss gauss.c –lm

I utilize these flags because they gave me better times than other, I tried some other flags, but I didn’t get good results.

➢ Justification for implementation choices

- **Swap rows implementation:** to swap the rows in partial pivoting I used the addresses of the rows in the double pointer matrix, and this was an efficient way to do it and better performance than swapping each value by row. Before this implementation, I swapped each value by row and my times were not that fast, so I thought about improving this because it wasn’t not an efficient way to it. So, storing the matrix with double pointers was one of the approaches I did to improve my timings and it gave better results, now my program doesn’t swap element by element in every row.

- **Back substitution:** I implemented back substitution as **column oriented** because I was having race conditions and performance was not efficient when I had it in row oriented. When I changed my code to column oriented, the race conditions were fixed because **loop-carried dependence** was some issue I had in the inner for loop of back substitution implementation for row oriented version that was causing race conditions. Switching to column oriented improved my performance as times went faster, so performance and race conditions were two important reasons I changed to column oriented. In this section of the code I used default to none for the scoped of variables inside the parallel directive **# omp parallel for**, and the reason was that inside this directive all variables declared inside it become private and outside the scope become shared by default, so I wanted to specify what variables were private and shared because this way I had more understanding of the code and I would know what to fix to prevent race conditions. Setting default scope to none was something I needed to get the correct result on back substitution and not produce race conditions.

- **Forward elimination:** As mentioned, I parallelized this section of code because each row would be assigned to every thread and would give better performance. I used default to none for the scope of variables because some of the variables needed to be shared and be private for all threads, for instance, the variable called **vec** is declared outside the scope of **# omp parallel for** and becomes shared by default and this is not something I wanted because this could lead to **race conditions**, and so setting it to private each **thread** gets its own private variable in its stack and prevents race conditions. Utilizing default to none help me exploit parallelism and synchronize threads to assign their result to the copy of the augmented matrix to produce the upper triangular after forward elimination.

➢ how successful your approach was, and what you would do differently or try next if you had more time!

- My approach was somewhat successful because some cores reached good efficiency and speedup. Cores from 1 to 20 reached a speedup close to the number of cores, like process 10 had a speedup of 9.4 which is close to 10 and core 5 reached a speedup of 4.8 and to reach perfect speedup the formula SU = T1 (time to complete task for one core) / Tp (time for processor to complete task) must be equal to p, and this is unlikely but these cores reflected a good speedup, however, cores from 30 to 60 reached an speedup for about 22, but it was not closed to the number of cores, and it is observed in the graph that speedup increases as number of cores increases. On the other hand, efficiency decreased as number of cores increased, and like speedup the less cores the better result, for example cores from 2 to 10 reached a good efficiency close to 1, and efficiency of 1 is considered perfect, but unlikely. However, overhead and communication between processes is something to be considered when number of cores increases and this might prevent perfect speedup and efficiency as observed in my results, for instance, core 60 reached and efficiency of .37 and speedup of 22.7. On the other hand, the column oriented approached I took to parallelize the function of back substitution gave good results as times went faster and **loop-carried dependence** was fixed in my code, and exploiting parallelism in the functions back substitution and forward elimination gave good results.
- What I would do if I had more time is to try to implement forward elimination more efficiently, because I think the code could be refactor, and I would try different techniques to parallelize my code more efficiently.

Table of minimum times from all runs and L2norm with minimum times highlighted with n = 8000

# core	1st run	2nd run	3rd run
1	1319.321202	1317.811006	1317.836663
2	670.976598	667.908342	669.224287
5	272.494903	272.320922	272.450031
10	139.234984	139.442500	139.458482
20	74.206943	74.113128	74.051583
30	57.762777	57.867812	58.647975
40	57.911769	58.137494	57.647826
50	57.526154	57.658639	57.520198
60	58.679937	58.028646	57.869521

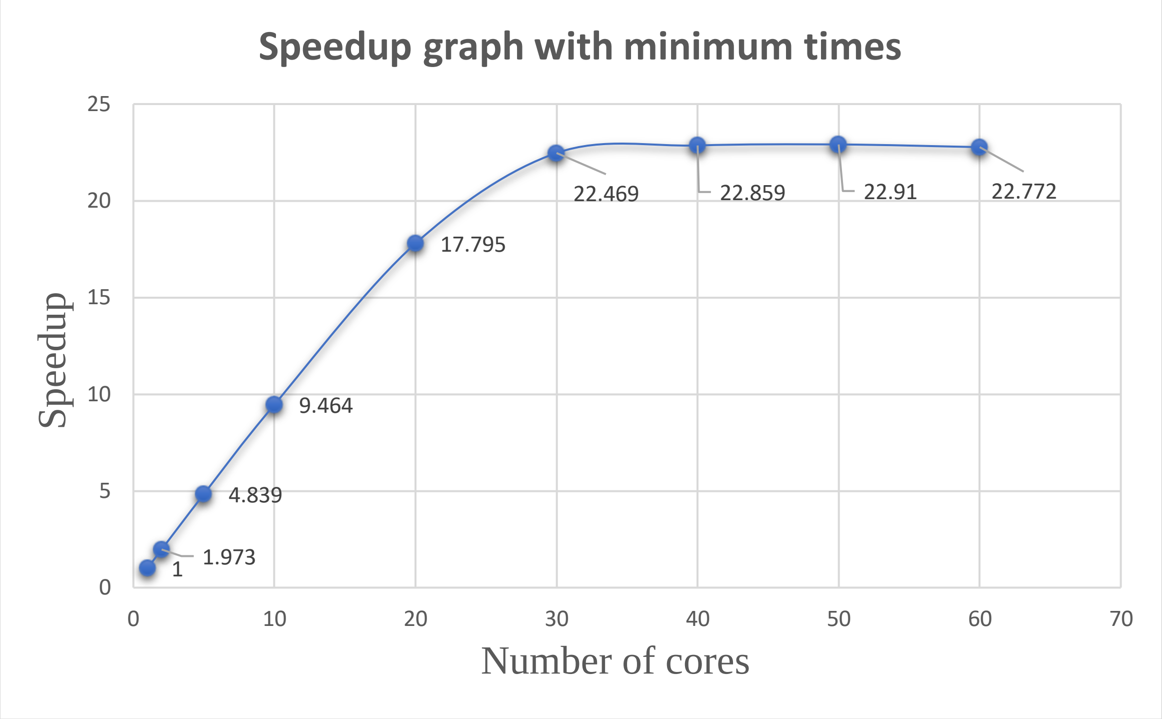
Table of L2norm for all runs with n = 8000

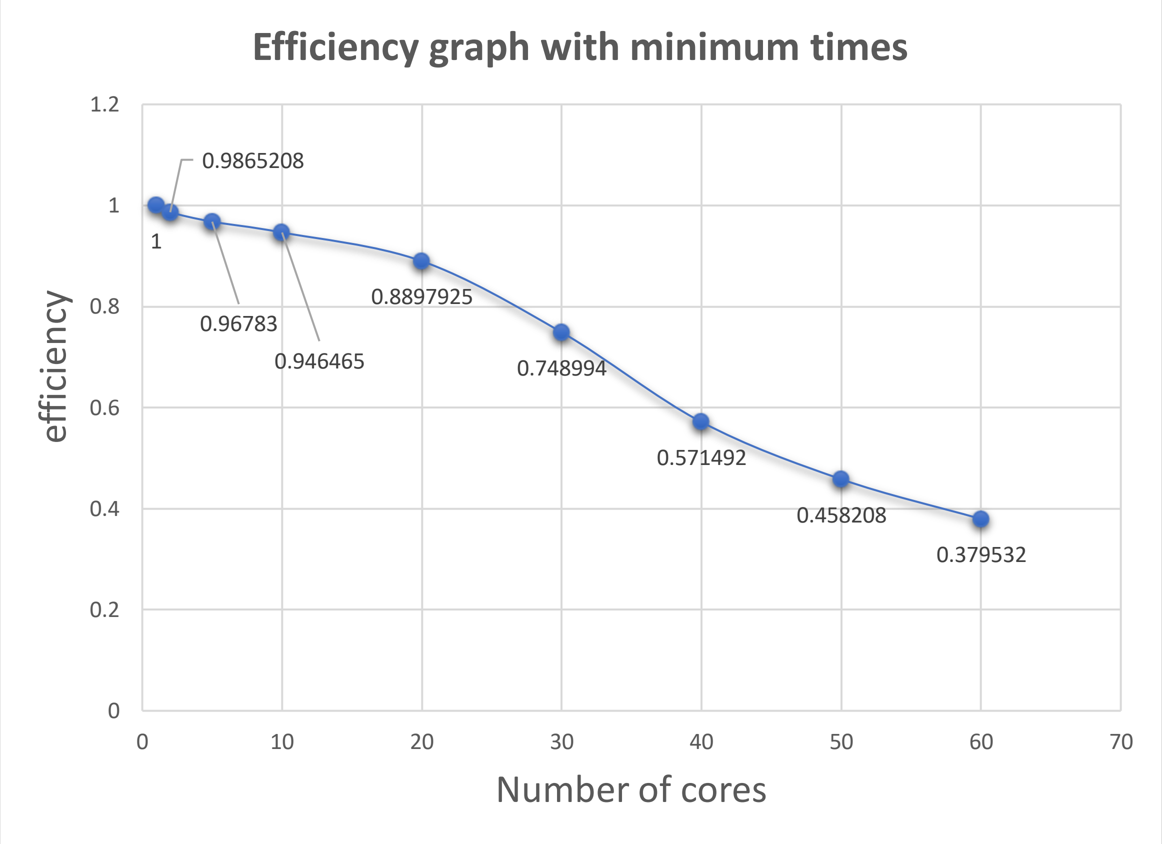
# core	1st run	2nd run	3rd run
1	1.8057077259e-03	9.9751062114e-04	4.5464521908e-02
2	7.4567364195e-04	1.6134919129e-03	4.3993668505e-03
5	1.9435404795e-03	7.2594779097e-04	3.1514652950e-03
10	9.7895652668e-04	5.9890855177e-02	1.7687183722e-03
20	1.1905188070e-03	2.0309855745e-03	5.8836505338e-03
30	2.8934600246e-03	1.3830714326e-03	1.6706940983e-02
40	3.5853718010e-03	8.5497383370e-04	1.4642857001e-03
50	2.1954802961e-03	4.0624714082e-03	2.2658849288e-03
60	2.7427772342e-03	7.9006755480e-03	2.7668743214e-03

Table of minimum times, with speedup and efficiency

# cores	**Minimum time**	**Speedup**	**Efficiency**	**L2 norm**
1	1317.811006	1	1	9.9751062114e-04
2	667.908342	1.9734041	.9865208	1.6134919129e-03
5	272.320922	4.839183	.96783	7.2594779097e-04
10	139.234984	9.464654	.946465	9.7895652668e-04
20	74.051583	17.795851	.8897925	5.8836505338e-03
30	58.647975	22.469846	.748994	1.6706940983e-02
40	57.647826	22.85968	.5714920	1.4642857001e-03
50	57.520198	22.910404	.458208	2.2658849288e-03
60	57.869521	22.77210	.379532	2.7668743214e-03

Separate line graphs of speedup and efficiency


Speedup graph with minimum times
Speedup
Number of cores


Efficiency graph with minimum times
efficiency
Number of cores