CSCI551

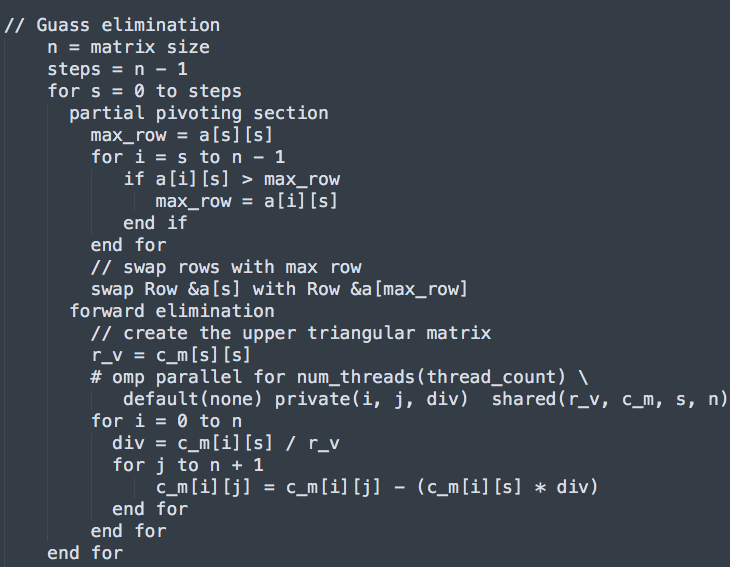
German Razo

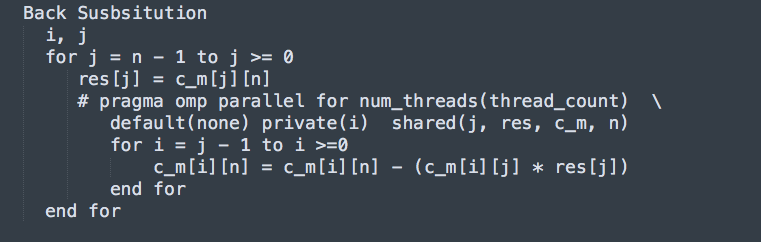
**Gaussian Elimination with OpenMP**

* **Data Storage**
* **Augmented matrix**: I stored the augmented matrix **A | b** in a double pointer array called **matrix** of size n, rows were allocated with size n + 1 because I saved both **A | b** in the same array, so A and b are combined. 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 mentioned before, so **c\_m** is a copy of the augmented matrix**.**
* **L2-norm:** To calculate the result of L2norm I used the original augmented matrix stored in the variable called **matrix**.
* **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, every thread contributes to perform the product of two rows to produce the resultant upper triangular matrix.
    - **Back substitution:** Inside this function I solved for back substitution, and I realized this 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 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 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 the section of the code waits for all thread to finish since there is an **implicit barrier**, so in this part is where synchronization happens because the **implicit barrier** ensures that all the threads have finished and all threads are join to the master thread after the block of omp parallel directive.

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. All threads are join to the master thread after the block that has the omp directive.

* **Pseudocode of key elements of the parallelization strategy**



****

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

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

* **Justification for implementation choices**
* **Swap rows implementation:** to swap the rows in partial pivoting I used the addresses of the rows in the 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. So, storing the matrix with double pointers was one of the approaches I did to improve my timings and it gave better results.
* **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 dependence was fixed in the inner for loop of back substitution implementation. 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.
* **how successful your approach was, and what you would do differently or try next if you had more time!**

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

|  |  |  |  |
| --- | --- | --- | --- |
| # core | 1st run | 2nd run | 3rd run |
| 1 |  |  |  |
| 2 |  |  |  |
| 5 |  |  |  |
| 10 |  |  |  |
| 20 |  |  |  |
| 30 |  |  |  |
| 40 |  |  |  |
| 50 |  |  |  |
| 60 |  |  |  |

L2-norms for each run

|  |  |  |  |
| --- | --- | --- | --- |
| # core | 1st run | 2nd run | 3rd run |
| 1 |  |  |  |
| 2 |  |  |  |
| 5 |  |  |  |
| 10 |  |  |  |
| 20 |  |  |  |
| 30 |  |  |  |
| 40 |  |  |  |
| 50 |  |  |  |
| 60 |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # cores | Minimum time | Speedup | Efficiency | L2 norm |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Speed up

Efficiency