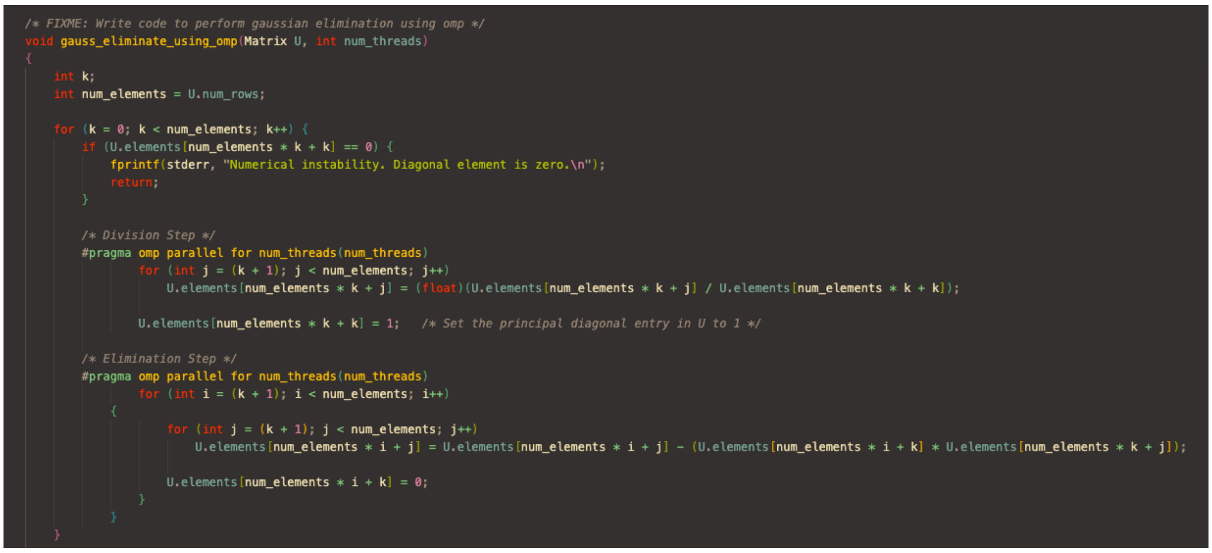
Ehi Simon - 14352307 ECEC 413-001
Alisha Augustin - 14389451 05/21/2023

Open MP: Gaussian Elimination

The figure below demonstrates the parallelization of the Gaussian elimination method using OpenMP. The *gauss_eliminate_using_omp* function takes as input the matrix *U* and the number of threads to use *num_threads*.The parallelization process begins by iterating over each row *k* of the matrix *U*. Within each iteration, the code checks if the diagonal element *U.elements[num_elements * k + k]* is zero. If so, it indicates a numerical instability, and an error message is printed.

The function then proceeds with the division step, which divides each element in the row *k+1* to num_elements by the diagonal element *U.elements[num_elements * k + k]*. This division step is parallelized using an OpenMP parallel loop. Each thread is assigned a portion of the iteration space, allowing concurrent computation. After completing the division step, the principal diagonal entry in U is set to 1. This ensures that the diagonal elements remain non-zero and avoids numerical instability. Next, the elimination step is performed, which subtracts the appropriate multiples of row *k* from subsequent rows to eliminate the entries below the diagonal. Similar to the division step, the elimination step is parallelized using an OpenMP parallel loop. Each thread handles a portion of the iteration space, resulting in concurrent execution. Finally, the loop moves to the next row *k+1* and repeats the process until all rows have been processed.



```c
/* FIXME: Write code to perform gaussian elimination using omp */
void gauss_eliminate_using_omp(Matrix U, int num_threads)
{
    int k;
    int num_elements = U.num_rows;

    for (k = 0; k < num_elements; k++) {
        if (U.elements[num_elements * k + k] == 0) {
            fprintf(stderr, "Numerical instability. Diagonal element is zero.\n");
            return;
        }

        /* Division Step */
        #pragma omp parallel for num_threads(num_threads)
                for (int j = (k + 1); j < num_elements; j++)
                    U.elements[num_elements * k + j] = (float)(U.elements[num_elements * k + j] / U.elements[num_elements * k + k]);

                U.elements[num_elements * k + k] = 1;   /* Set the principal diagonal entry in U to 1 */

        /* Elimination Step */
        #pragma omp parallel for num_threads(num_threads)
                for (int i = (k + 1); i < num_elements; i++)
            {

                    for (int j = (k + 1); j < num_elements; j++)
                        U.elements[num_elements * i + j] = U.elements[num_elements * i + j] - (U.elements[num_elements * i + k] * U.elements[num_elements * k + j]);

                    U.elements[num_elements * i + k] = 0;
                }
            }
    }
```

Figure 1: Implementation of Gaussian Elimination using OpenMP

Performance (Execution Time) Comparison

	Serial				**Parallel**			
Matrix Size	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
512x512	0.06s	0.06s	0.05s	0.05s	0.02s	0.03s	0.04s	0.11s
1024x1024	0.38s	0.38s	0.32s	0.30s	0.13s	0.12s	0.15s	0.23s
2048x2048	2.50s	2.57s	2.43s	2.44s	0.72s	0.49s	0.52s	0.68s
4096x4096	22.96s	21.35s	21.20s	22.0s	5.94s	4.41s	4.62s	4.99s

Speedup Comparison

Matrix Size	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
512x512	3	2	1.25	0.45
1024x1024	2.92	3.17	2.13	1.30
2048x2048	3.47	5.24	4.67	3.59
4096x4096	3.86	4.84	4.59	4.41

Generally, increasing the number of threads leads to improved speedup for larger matrix sizes. The speedup tends to be higher for smaller matrix sizes compared to larger ones. The speedup diminishes as the number of threads increases beyond a certain point, indicating diminishing returns in parallelization efficiency.