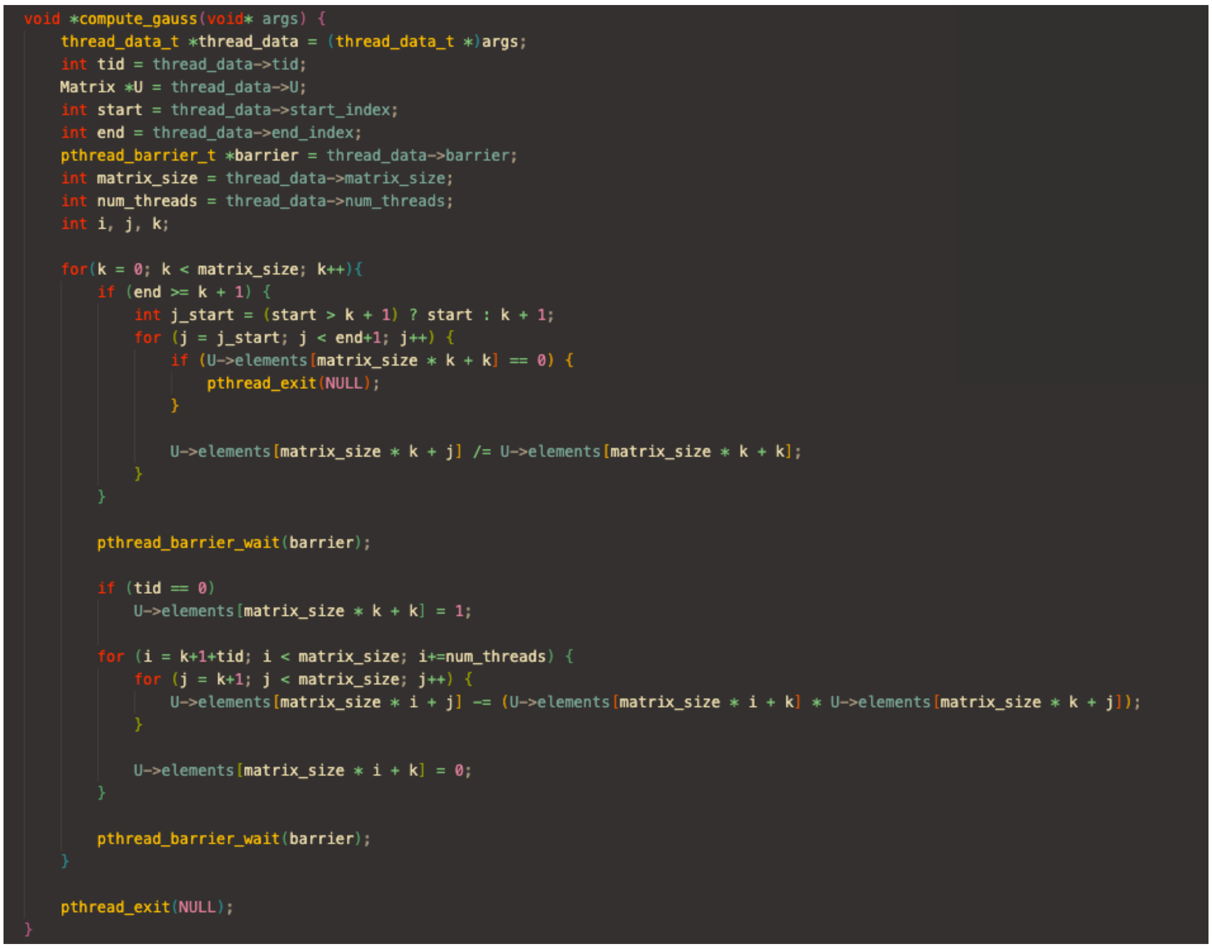
Ehi Simon - 14352307 ECEC 413-001

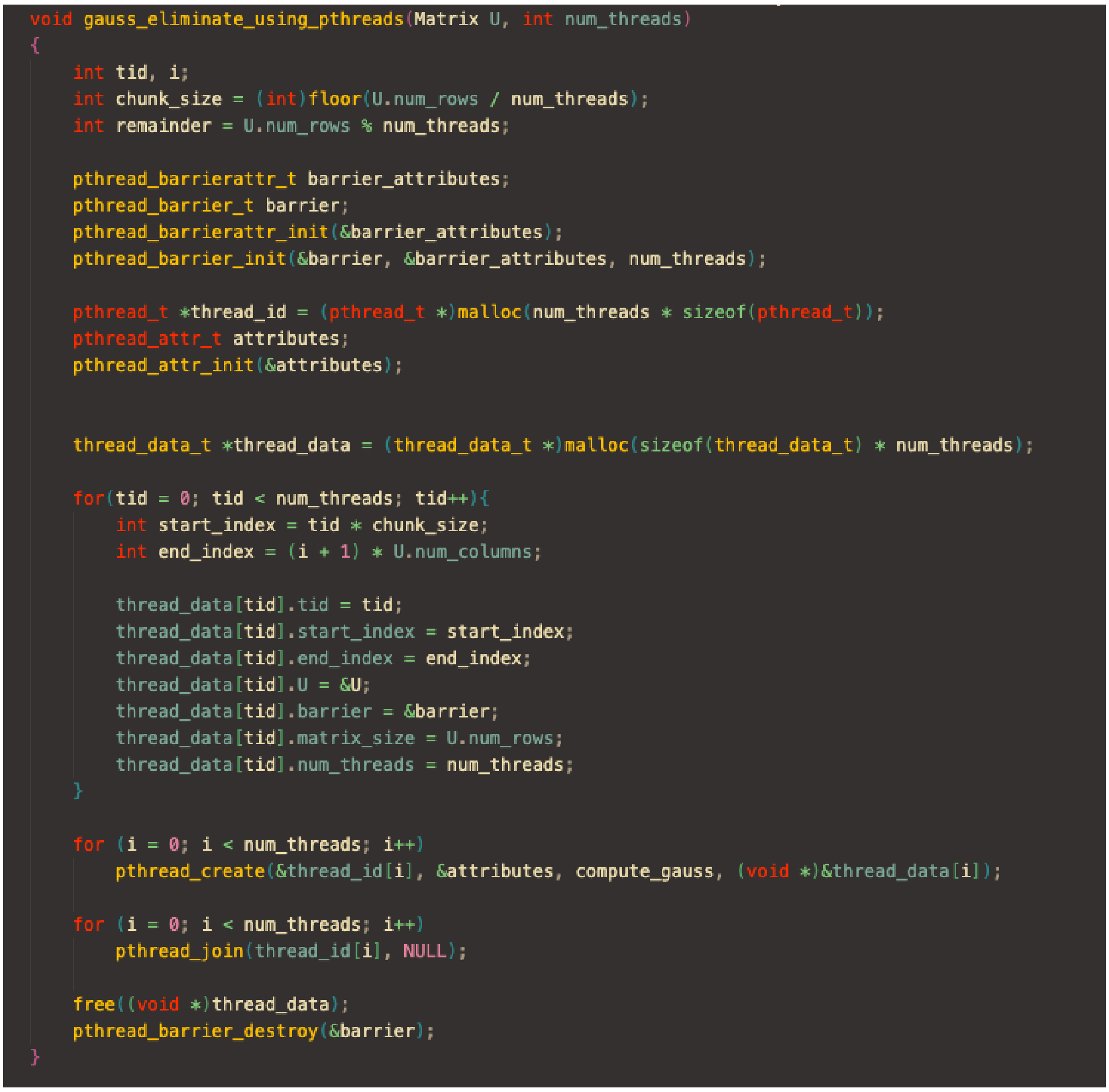
Alisha Augustin - 14389451 05/08/2023

Project 3: Gaussian Elimination

Gaussian elimination is a method for solving systems of linear equations and finding the reduced row-echelon form of a matrix. The gauss_eliminate_using_pthreads function is the main function that performs Gaussian elimination using multiple threads. It takes a Matrix object U as input and the number of threads num_threads to use for parallel computation. It divides the work among the threads and creates the necessary thread data structures. The function then creates the threads, assigns them the compute_gauss function to execute, and waits for all the threads to finish before cleaning up and exiting. The compute_gauss function is the thread function that performs the actual computation. The function extracts the necessary data from the args structure, including the thread ID, the matrix , the start and end indices for the portion of the matrix to be processed by the thread, the barrier object for synchronization, the matrix size, and the total number of threads. The thread then iterates over the rows of the matrix using the variable k and performs the Gaussian elimination steps. In each iteration, the thread checks if it is responsible for the current row (based on the start and end indices) and if the pivot element is zero. If the pivot is zero, the thread exits early. Next, the thread divides each element in the row by the pivot element to normalize it. Afterward, it synchronizes with other threads using the barrier to ensure that all threads have completed this step before moving on. The first thread sets the pivot element to 1 to maintain the upper triangular form. The thread then continues to update the rest of the matrix, subtracting multiples of the pivot row from the subsequent rows to eliminate the elements below the pivot. The loop is divided among the threads, with each thread handling a subset of rows based on its thread ID and the total number of threads. Finally, the thread synchronizes again using the barrier before moving on to the next iteration with a new pivot element. The code parallelizes the Gaussian elimination process by dividing the work among multiple threads, allowing for faster computation on systems with multiple cores or processors.



```c
void *compute_gauss(void* args) {
    thread_data_t *thread_data = (thread_data_t *)args;
    int tid = thread_data->tid;
    Matrix *U = thread_data->U;
    int start = thread_data->start_index;
    int end = thread_data->end_index;
    pthread_barrier_t *barrier = thread_data->barrier;
    int matrix_size = thread_data->matrix_size;
    int num_threads = thread_data->num_threads;
    int i, j, k;

    for(k = 0; k < matrix_size; k++){
        if (end >= k + 1) {
            int j_start = (start > k + 1) ? start : k + 1;
            for (j = j_start; j < end+1; j++) {
                if (U->elements[matrix_size * k + k] == 0) {
                    pthread_exit(NULL);
                }

                U->elements[matrix_size * k + j] /= U->elements[matrix_size * k + k];
            }
        }

        pthread_barrier_wait(barrier);

        if (tid == 0)
            U->elements[matrix_size * k + k] = 1;

        for (i = k+1+tid; i < matrix_size; i+=num_threads) {
            for (j = k+1; j < matrix_size; j++) {
                U->elements[matrix_size * i + j] -= (U->elements[matrix_size * i + k] * U->elements[matrix_size * k + j]);
            }

            U->elements[matrix_size * i + k] = 0;
        }

        pthread_barrier_wait(barrier);
    }

    pthread_exit(NULL);
}
```



```c
void gauss_eliminate_using_pthreads(Matrix U, int num_threads)
{
    int tid, i;
    int chunk_size = (int)floor(U.num_rows / num_threads);
    int remainder = U.num_rows % num_threads;

    pthread_barrierattr_t barrier_attributes;
    pthread_barrier_t barrier;
    pthread_barrierattr_init(&barrier_attributes);
    pthread_barrier_init(&barrier, &barrier_attributes, num_threads);

    pthread_t *thread_id = (pthread_t *)malloc(num_threads * sizeof(pthread_t));
    pthread_attr_t attributes;
    pthread_attr_init(&attributes);


    thread_data_t *thread_data = (thread_data_t *)malloc(sizeof(thread_data_t) * num_threads);

    for(tid = 0; tid < num_threads; tid++){
        int start_index = tid * chunk_size;
        int end_index = (i + 1) * U.num_columns;

        thread_data[tid].tid = tid;
        thread_data[tid].start_index = start_index;
        thread_data[tid].end_index = end_index;
        thread_data[tid].U = &U;
        thread_data[tid].barrier = &barrier;
        thread_data[tid].matrix_size = U.num_rows;
        thread_data[tid].num_threads = num_threads;
    }

    for (i = 0; i < num_threads; i++)
        pthread_create(&thread_id[i], &attributes, compute_gauss, (void *)&thread_data[i]);

    for (i = 0; i < num_threads; i++)
        pthread_join(thread_id[i], NULL);

    free((void *)thread_data);
    pthread_barrier_destroy(&barrier);
}
```

Performance 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	0.11s	0.11s	0.13s	0.21s
1024x1024	0.32s	0.30s	0.38s	0.38s	0.49s	0.46s	0.57s	0.73s
2048x2048	2.57s	2.57s	2.45s	2.78s	3.56s	3.16s	3.01s	3.62s
4096x4096	21.28s	21.94s	21.22s	21.74s	28.78s	29.37s	28.59s	29.13s

Speedup Comparison

Matrix Size	4 Threads	8 Threads	16 Threads	32 Threads
512x512	0.545	0.545	0.385s	0.0105
1024x1024	0.653	0.652	0.667	0.52
2048x2048	0.722	0.813	0.814	0.768
4096x4096	0.739	0.747	0.742	0.746

The speedup values indicate the performance improvement achieved by parallel execution compared to the serial execution. A speedup greater than 1 implies that the parallel execution is faster than the serial execution. For the 512x512 matrix size, the speedup values are consistently around 0.5 for 4 threads and 8 threads. However, for 16 threads, the speedup drops to 0.385, indicating diminishing returns. This suggests that the parallelization overhead might be impacting performance. For the 1024x1024 matrix size, the speedup values are slightly higher, ranging from 0.52 to 0.667. This indicates better parallel performance compared to the smaller matrix size. For the larger matrix sizes (2048x2048 and 4096x4096), the speedup values fluctuate but generally remain below 1. This suggests that the parallel execution may not be providing significant performance improvements for these sizes or that there are scalability limitations in the parallelization approach used. Overall, the speedup results show that the parallel execution does not consistently outperform the serial execution. It is important to analyze the factors contributing to this, such as overhead, resource contention, and scalability limitations, to identify potential areas for optimization and improve the parallel performance.