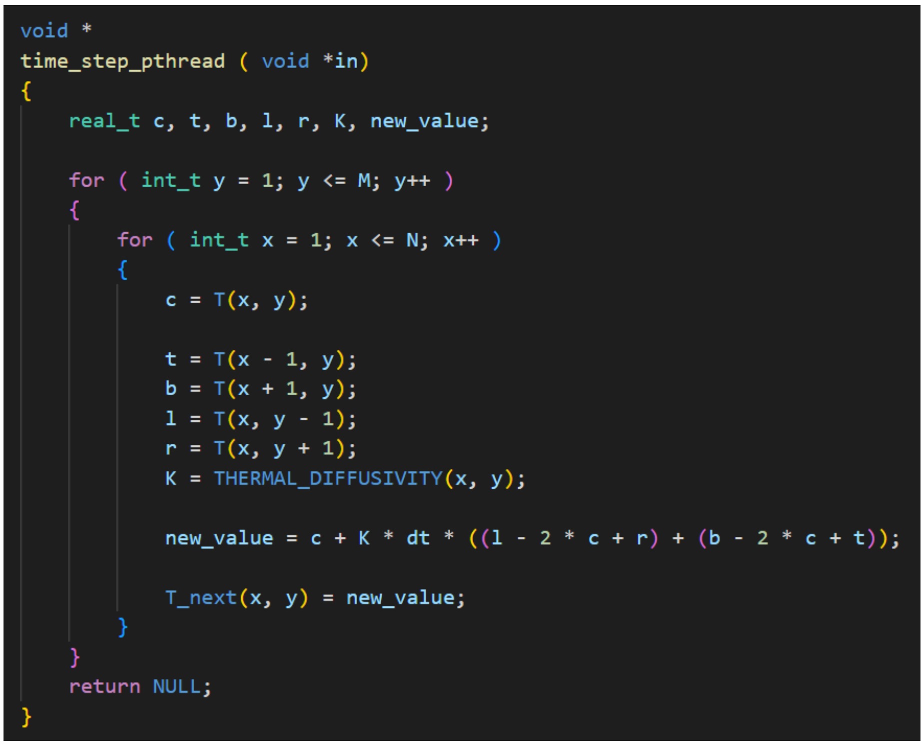
Exersice 4

Louka CHAPUS

1. Sequential Pthreads

To make the execution run faster we need to parallelise the time_step function because it's where we have the longest loop and where we do the actual calculation. So, when we have the function, we just need to create a pthread and join.

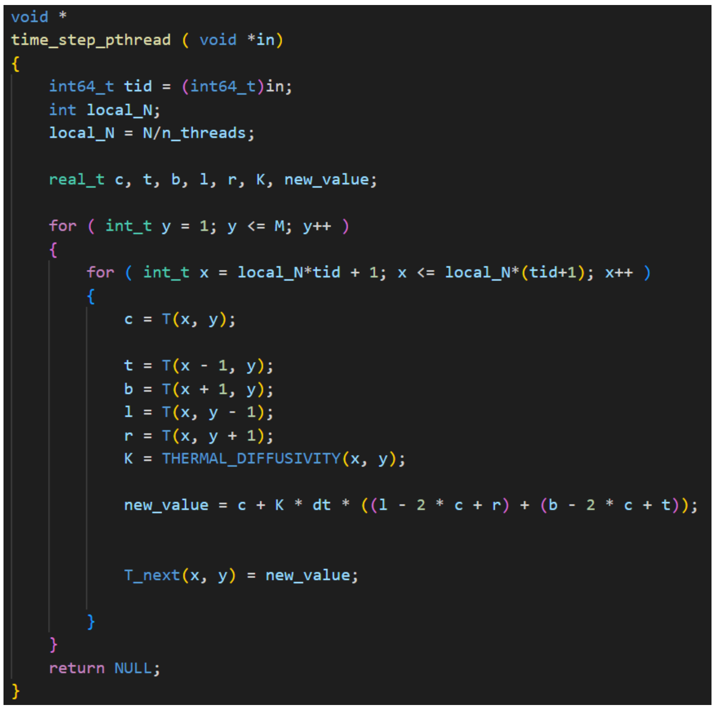


```c
void *
time_step_pthread ( void *in)
{
    real_t c, t, b, l, r, K, new_value;

    for ( int_t y = 1; y <= M; y++ )
    {
        for ( int_t x = 1; x <= N; x++ )
        {
            c = T(x, y);

            t = T(x - 1, y);
            b = T(x + 1, y);
            l = T(x, y - 1);
            r = T(x, y + 1);
            K = THERMAL_DIFFUSIVITY(x, y);

            new_value = c + K * dt * ((l - 2 * c + r) + (b - 2 * c + t));

            T_next(x, y) = new_value;
        }
    }
    return NULL;
}
```

Also, we need to return NULL because it's a void function and we don't use the argument because we have only one pthread for this part. The program takes 117 seconds to run on my computer.

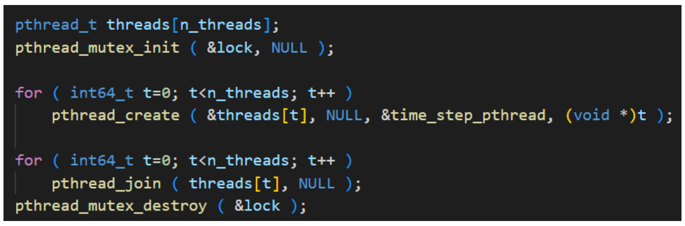
2. Parallel Pthreads

To run the program faster we can make multiple pthreads, so it does the calculation in parallel, and we also need to adapt the working space for each pthreads.



```c
void *
time_step_pthread ( void *in)
{
    int64_t tid = (int64_t)in;
    int local_N;
    local_N = N/n_threads;

    real_t c, t, b, l, r, K, new_value;

    for ( int_t y = 1; y <= M; y++ )
    {
        for ( int_t x = local_N*tid + 1; x <= local_N*(tid+1); x++ )
        {
            c = T(x, y);

            t = T(x - 1, y);
            b = T(x + 1, y);
            l = T(x, y - 1);
            r = T(x, y + 1);
            K = THERMAL_DIFFUSIVITY(x, y);

            new_value = c + K * dt * ((l - 2 * c + r) + (b - 2 * c + t));


            T_next(x, y) = new_value;

        }
    }
    return NULL;
}
```

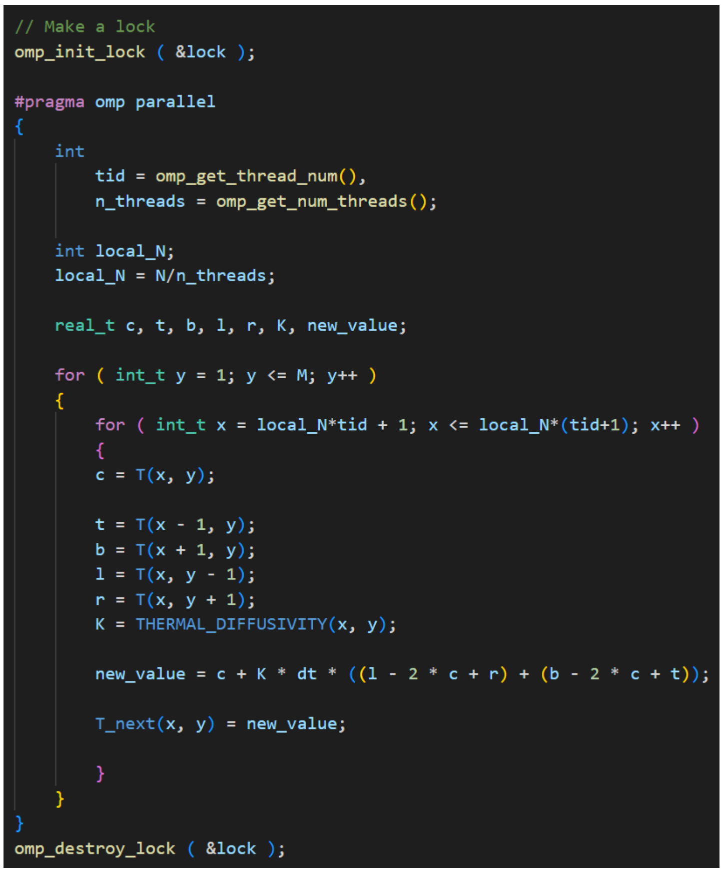
To spawn them and join them, we just use a loop in the main execution. The number of pthreads is define at the beginning of the program. Again, the execution took me 91 seconds to run. So, we have reduced the execution time by 25 seconds by making multiple pthreads and we still have the correct result.



```c
pthread_t threads[n_threads];
pthread_mutex_init ( &lock, NULL );

for ( int64_t t=0; t<n_threads; t++ )
    pthread_create ( &threads[t], NULL, &time_step_pthread, (void *)t );

for ( int64_t t=0; t<n_threads; t++ )
    pthread_join ( threads[t], NULL );
pthread_mutex_destroy ( &lock );
```

3. OpenMP

Next step, we use OpenMP to parallelize the execution with the pragma. So, we now implement the time_step function in the main execution. Again, we define the working space for each pthread with their id. We finally make a lock in between the pragma.



```
// Make a lock
omp_init_lock ( &lock );

#pragma omp parallel
{
    int
        tid = omp_get_thread_num(),
        n_threads = omp_get_num_threads();

    int local_N;
    local_N = N/n_threads;

    real_t c, t, b, l, r, K, new_value;

    for ( int_t y = 1; y <= M; y++ )
    {
        for ( int_t x = local_N*tid + 1; x <= local_N*(tid+1); x++ )
        {
        c = T(x, y);

        t = T(x - 1, y);
        b = T(x + 1, y);
        l = T(x, y - 1);
        r = T(x, y + 1);
        K = THERMAL_DIFFUSIVITY(x, y);

        new_value = c + K * dt * ((l - 2 * c + r) + (b - 2 * c + t));

        T_next(x, y) = new_value;

        }
    }
}
omp_destroy_lock ( &lock );
```

With this implementation the program runs faster than the previous implementation, it finishes the execution after 53 seconds on my computer, that's two time faster than the sequential implementation.

4. Questions

A critical section it's a part of the code who do a lot of execution (essentially a loop).

To protect a critical section from a race condition we can add some lock so each pthread write the data at a time.

Oversubscription is when a variable is overwritten in multiple treads.

To make sure that the number come in order we can make a lock and then wait for each thread to print the correct number.