```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <assert.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    int N, NT, nt;
    double L, T, u, v;
    N = atoi(argv[1]);
    NT = atoi(argv[2]);
    L = atof(argv[3]);
    T = atof(argv[4]);
    u = atof(argv[5]);
    v = atof(argv[6]);

    // The number of threads
    nt = atoi(argv[7]);

    // The method to use: 1 for Lax, 2 for first order method, 3 for second order
method
    int method = atoi(argv[8]);
    double t1, t2;

    omp_set_num_threads(nt);

    // Printing the inputs given by the user
    printf("The value of Matrix Dimension, N is:                %i\n", N);
    printf("The value of Number of timestamps, NT is:           %i\n", NT);
    printf("The value of Physical Cartesian Domain Length, L is:  %f\n", L);
    printf("The value of Total Physical Timespan, T is:         %f\n", T);
    printf("The value of X velocity scalar, u is:               %.10f\n", u);
    printf("The value of Y velocity scalar, v is:               %.10f\n", v);
    printf("The number of threads, nt is/are:                   %i\n", nt);

    int mem = N * N * 2;
    printf("Memory to be used will be %d time size of double\n", mem);

    // Creating a 2D array using pointers
    double **Cn = (double**)malloc(N * sizeof(double*));
    double **Cn1 = (double**)malloc(N * sizeof(double*));
    double **temp;

    for (int i = 0; i < N; i++)
    {
        Cn1[i] = (double*)malloc(N * sizeof(double));
```

```
        Cn[i] = (double*)malloc(N * sizeof(double));
    }

    double delx = L / N;
    double delt = T / NT;

    assert(delt <= delx / sqrt(2 * (pow(u, 2) + pow(v, 2))));

    double x0 = L / 2;
    double y0 = x0;

    double sigx = L / 4;
    double sigy = sigx;

    // Initialising Cn using 2D Gaussian pulse initial condition
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {

            Cn[i][j] = exp(
                -(
                    pow(i * delx - x0, 2) / (2 * pow(sigx, 2)) +
                    pow(j * delx - y0 ,2) / (2 * pow(sigy, 2))
                )
            );
        }

    t1 = omp_get_wtime();

    for (int n = 0; n < NT; n++)
    {
        #ifdef PARALLEL
        #pragma omp parallel for default(none) shared(N, NT, L, T, u, v, method, Cn,
Cn1, temp, delx, delt, x0, y0, sigx, sigy, n) schedule(static)
        #endif
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                // The boundary conditions. Wrapping around whenever encountering a
boundary
                int im = i - 1;
                int imm = i - 2;
                int ip = i + 1;
                int ipp = i + 2;
                int jm = j - 1;
                int jmm = j - 2;
                int jp = j + 1;
```

```
                int jpp = j + 2;
                if (i == 0)
                {
                    im = N - 1;
                    imm = N - 2;
                }
                if (j == 0)
                {
                    jm = N - 1;
                    jmm = N - 2;
                }
                if (i == N - 1)
                {
                    ip = 0;
                    ipp = 1;
                }
                if (j == N - 1)
                {
                    jp = 0;
                    jpp = 1;
                }
                if (i == 1)
                    imm = N - 1;
                if (j == 1)
                    jmm = N - 1;
                if (i == N - 2)
                    ipp = 0;
                if (j == N - 2)
                    jpp = 0;

                // Updating Cn1

                // Lax Method
                if (method == 1)
                    Cn1[i][j] = 1.0 / 4.0 * (Cn[im][j] + Cn[ip][j] + Cn[i][jm] +
Cn[i][jp]) - delt / (2.0 * delx) * (u * (Cn[ip][j] - Cn[im][j]) + v * (Cn[i][jp] -
Cn[i][jm]));

                // First Order Method
                else if (method == 2)
                {
                    if (u > 0 && v > 0)
                        Cn1[i][j] = Cn[i][j] - delt / delx * (u * (Cn[i][j] -
Cn[im][j]) + v * (Cn[i][j] - Cn[i][jm]));
                    else
                        Cn1[i][j] = Cn[i][j] - delt / delx * (u * (Cn[ip][j] -
Cn[i][j]) + v * (Cn[i][jp] - Cn[i][j]));
                }
```

```
                // Second Order Method
                else if (method == 3)
                {
                    if (u > 0 && v > 0)
                        Cn1[i][j] = Cn[i][j] - delt / (2.0 * delx) * (u * (3.0 *
Cn[i][j] - 4.0 * Cn[im][j] + Cn[imm][j]) + (v * (3.0 * Cn[i][j] - 4.0 * Cn[i][jm] +
Cn[i][jmm])));
                    else
                        Cn1[i][j] = Cn[i][j] + delt / (2.0 * delx) * (u * (3.0 *
Cn[i][j] - 4.0 * Cn[ip][j] + Cn[ipp][j]) + (v * (3.0 * Cn[i][j] - 4.0 * Cn[i][jp] +
Cn[i][jpp])));
                }
            }
        }

        // Storing value of Cn1 to Cn. Swapping it to ensure that we dont lose
reference to the pointer of Cn1
        temp = Cn1;
        Cn1 = Cn;
        Cn = temp;
    }

    t2 = omp_get_wtime();
    printf("time(s): %f\n", t2 - t1);

    // Freeing up memory
    for (int i = 0; i < N; i++)
    {
        free(Cn[i]);
        free(Cn1[i]);
    }

    free(Cn);
    free(Cn1);
    return 0;
}
```