navier_stokes_paralelo_spread.c

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

// NT agora é uma constante interna.
#define NT 2000

// Funções auxiliares agora recebem as dimensões como parâmetros
double** allocate_grid(int nx, int ny) {
    double *data = (double*)malloc(nx * ny * sizeof(double));
    double **array = (double**)malloc(nx * sizeof(double*));
    for (int i = 0; i < nx; i++) {
        array[i] = &(data[i * ny]);
    }
    return array;
}

void free_grid(double** array) {
    free(array[0]);
    free(array);
}

int main(int argc, char *argv[]) {
    // Agora esperamos o tamanho da grade (NX) como argumento
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <TAMANHO_DA_GRADE>\n", argv[0]);
        fprintf(stderr, "Exemplo: %s 512\n", argv[0]);
        return 1;
    }
    int NX = atoi(argv[1]);
    int NY = NX; // NY será sempre igual a NX

    // Alocação de memória usa as variáveis NX e NY
    double **u = allocate_grid(NX, NY);
    double **v = allocate_grid(NX, NY);
    double **u_new = allocate_grid(NX, NY);
    double **v_new = allocate_grid(NX, NY);

    // Inicialização usa as variáveis NX e NY
    #pragma omp parallel for
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            u[i][j] = 1.0; v[i][j] = 0.0;
            double dx = i - NX/2.0, dy = j - NY/2.0;
            double dist = sqrt(dx*dx + dy*dy);
            if (dist < (NX / 25.0)) { // Condição inicial relativa ao tamanho
da grade
                u[i][j] += 2.0 * exp(-dist*dist/(NX/5.0));
                v[i][j] += 1.5 * exp(-dist*dist/(NX/5.0));
```

```
            }
        }
    }

    double start_time = omp_get_wtime();

    #pragma omp parallel proc_bind(spread)
    {
        for (int step = 0; step < NT; step++) {

            #pragma omp for collapse(2) schedule(static)
            for (int i = 1; i < NX-1; i++) {
                for (int j = 1; j < NY-1; j++) {
                    double d2u_dx2 = (u[i+1][j] - 2.0*u[i][j] + u[i-1][j]);
                    double d2u_dy2 = (u[i][j+1] - 2.0*u[i][j] + u[i][j-1]);
                    double d2v_dx2 = (v[i+1][j] - 2.0*v[i][j] + v[i-1][j]);
                    double d2v_dy2 = (v[i][j+1] - 2.0*v[i][j] + v[i-1][j]);

                    u_new[i][j] = u[i][j] + (0.001 * 0.01) * (d2u_dx2 +
d2u_dy2); // DT e NU podem ser fixados
                    v_new[i][j] = v[i][j] + (0.001 * 0.01) * (d2v_dx2 +
d2v_dy2);
                }
            }

            #pragma omp for
            for (int i = 0; i < NX; i++) {
                u_new[i][0] = u_new[i][NY-2];
                u_new[i][NY-1] = u_new[i][1];
                v_new[i][0] = v_new[i][NY-2];
                v_new[i][NY-1] = v_new[i][1];
            }

            #pragma omp for
            for (int j = 0; j < NY; j++) {
                u_new[0][j] = u_new[NX-2][j];
                u_new[NX-1][j] = u_new[1][j];
                v_new[0][j] = v_new[NX-2][j];
                v_new[NX-1][j] = v_new[1][j];
            }

            #pragma omp single
            {
                double **temp_u = u;
                double **temp_v = v;
                u = u_new;
                v = v_new;
                u_new = temp_u;
                v_new = temp_v;
            }
        }
    }

```

```
101     double end_time = omp_get_wtime();
102     printf("%.6f\n", end_time - start_time);
103 
104     free_grid(u); free_grid(v); free_grid(u_new); free_grid(v_new);
105 
106     return 0;
107 }
108 
```