**paralelo/navier_stokes_simul_paralela_otm_st.c**

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

#define NX 512
#define NY 512
#define NT 10000 // Mantendo o NT alto para testes de desempenho
#define DT 0.001
#define NU 0.01

int main() {
    // Alocar memória (sem alterações)
    double **u = malloc(NX * sizeof(double*));
    double **v = malloc(NX * sizeof(double*));
    double **un = malloc(NX * sizeof(double*));
    double **vn = malloc(NX * sizeof(double*));

    for (int i = 0; i < NX; i++) {
        u[i] = malloc(NY * sizeof(double));
        v[i] = malloc(NY * sizeof(double));
        un[i] = malloc(NY * sizeof(double));
        vn[i] = malloc(NY * sizeof(double));
    }

    // Inicialização (pode ser otimizada com collapse também)
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            double dx = i - NX/2, dy = j - NY/2;
            double dist_sq = dx*dx + dy*dy;

            u[i][j] = 1.0;
            v[i][j] = 0.0;

            if (dist_sq < 400) {
                double perturbation = exp(-dist_sq/100.0);
                u[i][j] += 2.0 * perturbation;
                v[i][j] += 1.5 * perturbation;
            }
        }
    }

    double start = omp_get_wtime();

    // Loop de tempo PRINCIPAL - continua serial
    for (int t = 0; t < NT; t++) {

        // OTIMIZAÇÃO 1: UMA ÚNICA REGIÃO PARALELA
        // As threads são criadas apenas uma vez por passo de tempo.
```

```
51          #pragma omp parallel
52          {
53              // OTIMIZAÇÃO 2: COLLAPSE E SCHEDULE
54              // A cláusula collapse(2) trata os laços 'i' e 'j' como um único
    laço,
55              // melhorando muito o balanceamento de carga.
56              // A cláusula schedule controla como as iterações são distribuídas.
57              #pragma omp for collapse(2) schedule(static) /* Mude aqui para seus
    testes: (dynamic), (guided), (static, 64), etc. */
58              for (int i = 1; i < NX-1; i++) {
59                  for (int j = 1; j < NY-1; j++) {
60                      un[i][j] = u[i][j] + DT*NU*(u[i+1][j] + u[i-1][j] + u[i]
    [j+1] + u[i][j-1] - 4*u[i][j]);
61                      vn[i][j] = v[i][j] + DT*NU*(v[i+1][j] + v[i-1][j] + v[i]
    [j+1] + v[i][j-1] - 4*v[i][j]);
62                  }
63              }
64
65              // Condições de contorno dentro da mesma região paralela
66              // O omp for possui uma barreira implícita no final, garantindo que
    o cálculo
67              // principal termine antes de aplicar as condições de contorno.
68              #pragma omp for
69              for (int i = 0; i < NX; i++) {
70                  un[i][0] = un[i][NY-2];
71                  un[i][NY-1] = un[i][1];
72                  vn[i][0] = vn[i][NY-2];
73                  vn[i][NY-1] = vn[i][1];
74              }
75
76              #pragma omp for
77              for (int j = 0; j < NY; j++) {
78                  un[0][j] = un[NX-2][j];
79                  un[NX-1][j] = un[1][j];
80                  vn[0][j] = vn[NX-2][j];
81                  vn[NX-1][j] = vn[1][j];
82              }
83          } // Fim da região paralela. As threads são sincronizadas aqui.
84
85          // Swap pointers (feito pelo thread mestre, serialmente)
86          double **ut = u, **vt = v;
87          u = un; v = vn;
88          un = ut; vn = vt;
89      }
90
91      double end = omp_get_wtime();
92      printf("%.6f\n", end - start);
93
94      // Cleanup
95      for (int i = 0; i < NX; i++) {
96          free(u[i]); free(v[i]); free(un[i]); free(vn[i]);
97      }
98      free(u); free(v); free(un); free(vn);
```

```
 99
100     return 0;
101 }
```