NOTA: aqui, neste guião, ao contrário do que fazíamos na PL4 vamos utilizar (no terminal):

export OMP_NUM_THREADS = (N)

onde aqui colocaremos o nº de threads que pretendemos

Não precisamos de estar sempre a escrever isto, isto é, podemos escrever apenas quando queremos mudar o nº de threads.

PL5 → Exercício 1

```
#include <omp.h>
#include <stdio.h>

int main() {
    int w = 10;                          // é um simples contador que vai incrementar sempre que termina um ciclo e enquanto o i<100
#pragma omp parallel @2
#pragma omp for @3
    for (int i=0; i<100; i++) {          // @1
        int id = omp_get_thread_num();
        printf("T%d: i %d w=%d\n", id, i, w++);
    }
    printf("w=%d\n", w);
}
```

(variável w será declarada fora da região paralela, logo está a ser partilhada por todos os threads.)

@1 → As variáveis que estão dentro da região paralela vão ao variáveis privadas. Portanto, a variável id vai ser uma variável privada para as threads, ou seja, varia de acordo com cada thread.

@2 → como colocamos aquilo no terminal, neste pragma não há a necessidade de haver a indicação do nº de fios que vão ser usados.

@3 → como vimos na PL4, isto faz com que haja distribuição da carga (ex.: se N=4, então cada thread fica com 25).

→

Mas o "output" depende da thread que chegar 1º à região paralela.

→ estamos a usar 4 threads

↳ Se mandarmos executar o código assim, vamos obter o seguinte: → ver valor + pequeno!!

- valor inicial do w: $w=10$
- valor final do w dentro do loop: $w=108$ → mas se fizermos várias execuções, o valor dá sempre diferente

temos que pegar no último valor do w que aparece e subtrair 1

↳ Vamos retirar o #pragma omp for e colocar o #pragma omp for private(w). Mandando executar o código vamos ter que:

- valor inicial do w: $w=0$

o w é sempre inicializado a 0 para cada thread e tem sempre os mesmos valores (0-24 no caso de N=4).

↓

portanto, a carga vai estar igualmente distribuída por cada thread.

→ isto deve-se ao nº de iterações de cada fio

- valor final do w: $w=10$

para todas as execuções, w retorna o seu valor original

↓

ñ podemos identificar w como uma variável privada, pois aqui ñ há comunicação entre fios!!

↳ Vamos agora retirar o #pragma omp for private(w) e colocar o #pragma omp for firstprivate(w). Mandando executar o código vamos ter que:

- valor inicial do w: $w=10$ → que era o valor esperado!!

basicamente, ele vai pegar na nossa inicialização e colocar como o seu primeiro também

De resto, vai ser tudo igual ao pragma anterior, isto é, o w final é = a 10 na mesma e threads ≠ vão continuar a calcular os mesmos valores de w.

↳ Substituindo o firstprivate(w) por lastprivate(w) e mandando executar o código vamos ter o seguinte:

- valor inicial do w: w = 0

inicializamos da mesma forma que o "private"

- valor final do w: w = 25

que chegou a ser cumprido

é o último w calculado na última thread

(daí somando 1 porque na última thread w = 24 e o valor retornado é 25)

→ no final do ciclo, o w é sempre incrementado

↳ Por fim, vamos ter o * pragma omp for reduction(+:w) e mandando executar o código vamos ter que:

- valor inicial do w: w = 0
- valor final do w: w = 110 → valor correto!!

Cada fio vai executar o seu valor de w e no fim vamos ter a soma (porque é o que diz dentro do ()) de todos os valores que vai dar 100. Mas como o valor inicial é = a 10, então temos que somar isso também, sendo que temos w = 110!

⇓

este valor vai ser sempre = independentemente do nº de threads usadas e da quantidade de vezes que mandamos executar o código