

# Memorando

**De:** Willfredy Vieira Dias

**Nº de Matrícula:** 20200204

**Disciplina:** Computação Paralela e Distribuída (CPD)

**Assunto:** Aula de Laboratório Nº 2 – Introduzir o OpenMP

**Data:** 04/04/2025

## 1. Introdução

A experiência desenvolvida baseia-se em duas partes principais que visam introduzir os fundamentos do OpenMP:

**Contexto do Trabalho:** Realizado no âmbito da disciplina de Computação Paralela e Distribuída, durante a Aula de Laboratório nº 2 no ISPTEC 2021-22. O exercício tem como objetivo familiarizar os estudantes com o ambiente de compilação e execução de programas utilizando OpenMP.

### Objectivos:

- Apresentar e utilizar a API e as diretivas do OpenMP.
- Experimentar a execução paralela dos códigos, ajustando o número de threads através da variável de ambiente (OMP\_NUM\_THREADS).
- Analisar o impacto de cláusulas como “nowait” e a inserção de barreiras, evidenciando os efeitos da sincronização no comportamento e na saída do programa.
- Abordar a paralelização de um loop com dependências de dados, comparando uma abordagem inicial (incorreta) com uma versão corrigida, que preserva a integridade dos dados, e explorar alternativas para evitar a cópia desnecessária do vector.

### Restrições e Observações Importantes:

- É imprescindível compilar os códigos com a flag “-fopenmp” para ativar o suporte ao OpenMP.
- Recomenda-se a consulta à documentação oficial do OpenMP (versão 4.5) para esclarecer eventuais dúvidas sobre as directivas ou APIs.

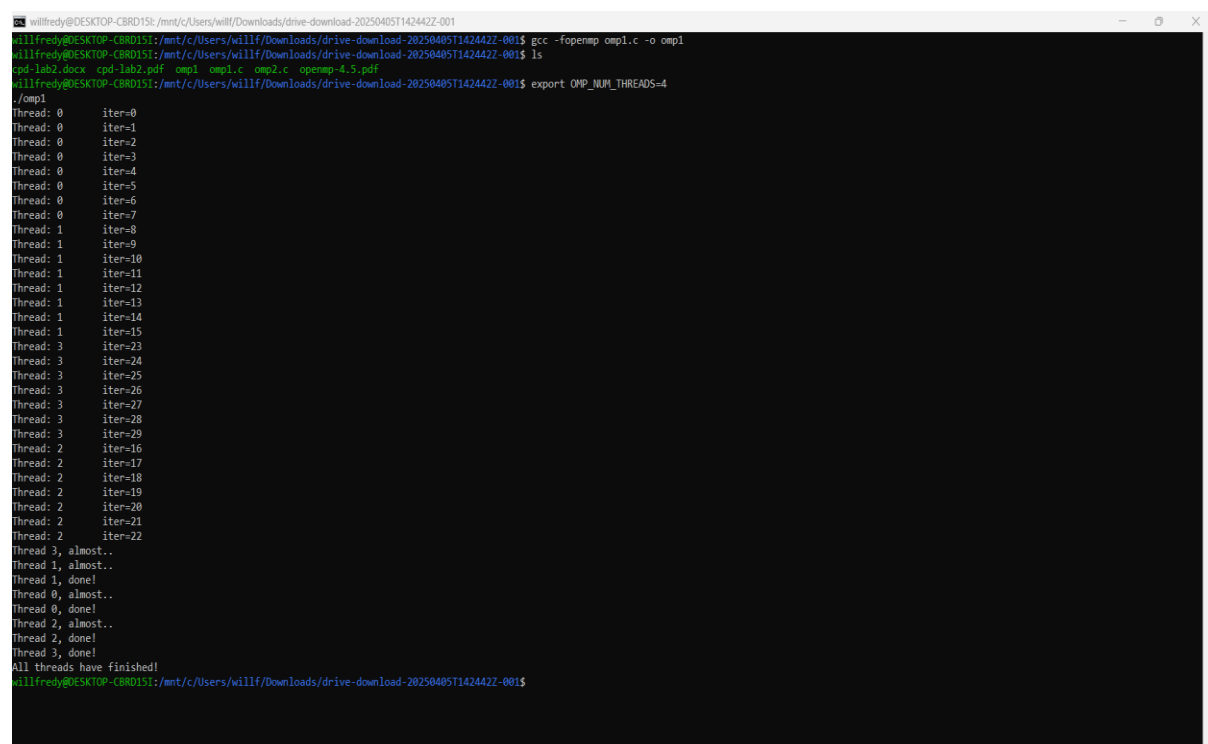
- No primeiro problema, a alteração da saída do programa ao adicionar a cláusula “nowait” e a barreira deve ser cuidadosamente analisada, uma vez que estas modificações afetam a sincronização entre as threads.
- No segundo problema, a paralelização do loop interno deve ser tratada com atenção, dada a existência de dependências de dados; uma implementação simples pode conduzir a resultados incoerentes, pelo que se torna necessário copiar os dados da iteração anterior ou adoptar estratégias que evitem esta cópia, mantendo assim a consistência do processamento.

Esta atividade permite uma compreensão prática dos desafios e das estratégias necessárias para implementar paralelismo, enfatizando a importância da sincronização e do correto tratamento das dependências de dados.

## 2. Experiências Realizadas

### Problema 1

a) Comece por compilar o código acima, não esqueça de adicionar a flag de compilação “-fopenmp”. Execute este programa com um número diferente de threads (definindo `OMP_NUM_THREADS`).



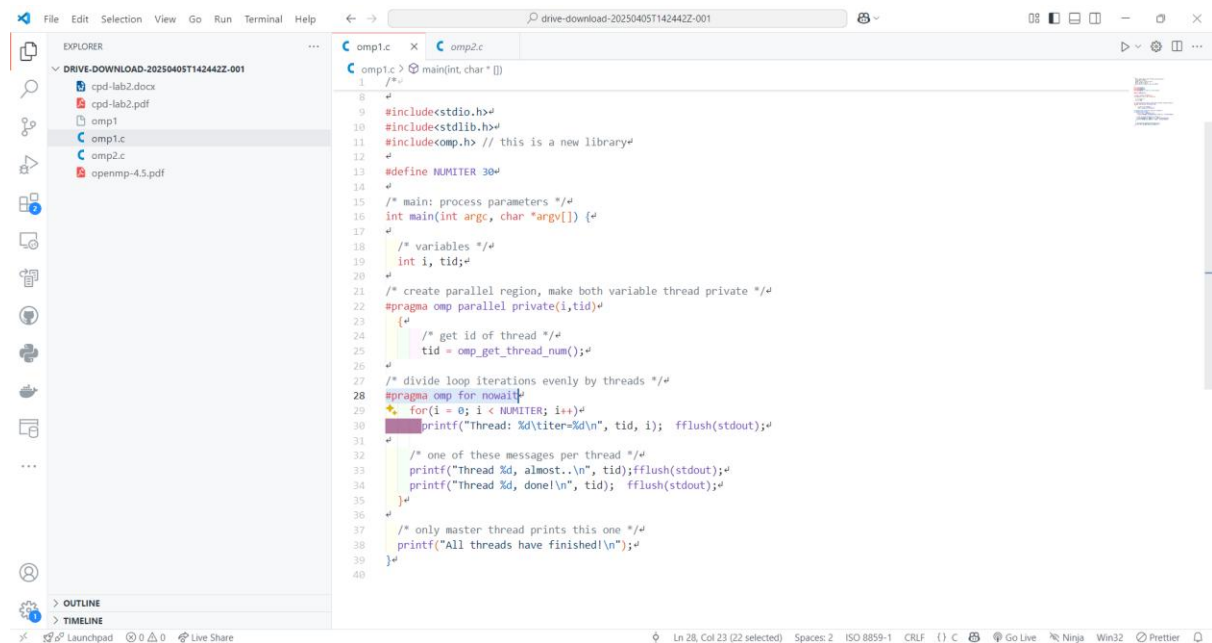
```

willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-20250405T142442Z-001
willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-20250405T142442Z-001$ gcc -fopenmp omp1.c -o omp1
willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-20250405T142442Z-001$ ls
cpd-1ab2.docx  cpd-1ab2.pdf  omp1  omp1.c  omp2.c  openmp-4.5.pdf
willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-20250405T142442Z-001$ export OMP_NUM_THREADS=4
./omp1
Thread: 0      iter=0
Thread: 0      iter=1
Thread: 0      iter=2
Thread: 0      iter=3
Thread: 0      iter=4
Thread: 0      iter=5
Thread: 0      iter=6
Thread: 0      iter=7
Thread: 1      iter=8
Thread: 1      iter=9
Thread: 1      iter=10
Thread: 1      iter=11
Thread: 1      iter=12
Thread: 1      iter=13
Thread: 1      iter=14
Thread: 1      iter=15
Thread: 3      iter=23
Thread: 3      iter=24
Thread: 3      iter=25
Thread: 3      iter=26
Thread: 3      iter=27
Thread: 3      iter=28
Thread: 3      iter=29
Thread: 2      iter=16
Thread: 2      iter=17
Thread: 2      iter=18
Thread: 2      iter=19
Thread: 2      iter=20
Thread: 2      iter=21
Thread: 2      iter=22
Thread: 3, almost..
Thread: 1, almost..
Thread: 1, done!
Thread: 0, almost..
Thread: 0, done!
Thread: 2, almost..
Thread: 2, done!
Thread: 3, done!
All threads have finished!
willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-20250405T142442Z-001$

```

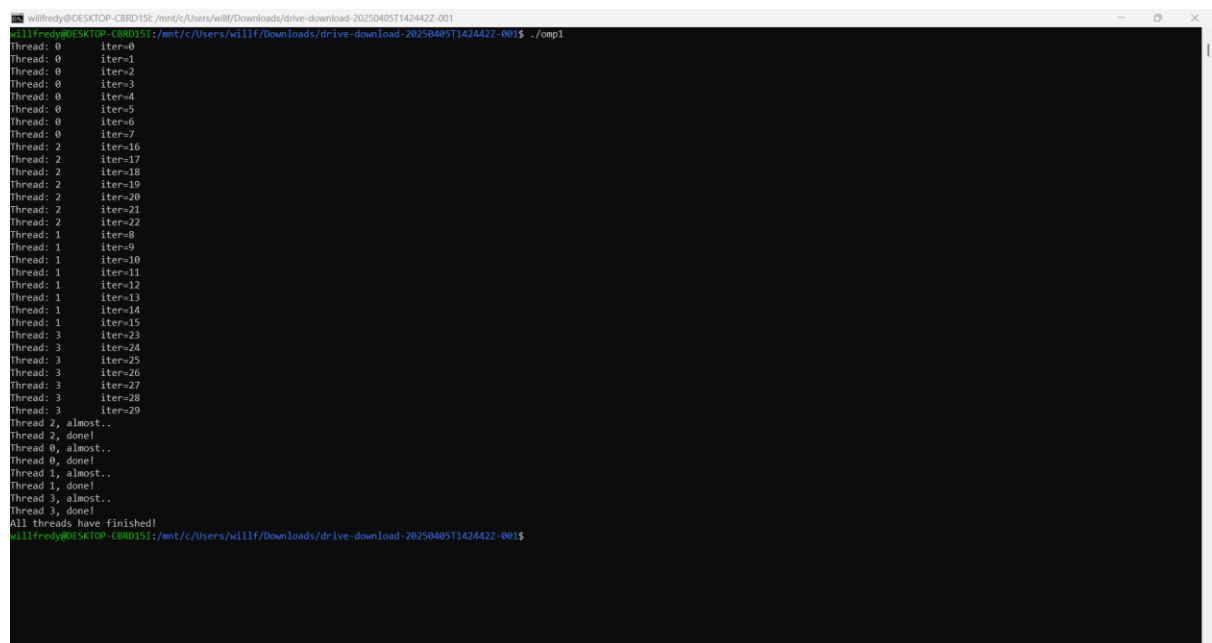
figura a 1 - Execução do programa com o `OMP_NUM_THREADS`.

**b) Adicione a cláusula “*nowait*” à directiva **for** na linha 9. A saída foi alterada? Porquê?**



```
1  /*  
2  
3  
4  
5  
6  
7  
8  
9  #include<stdio.h>  
10 #include<stdlib.h>  
11 #include<omp.h> // this is a new library  
12  
13 #define NUMITER 30  
14  
15 /* main: process parameters */  
16 int main(int argc, char *argv[]) {  
17  
18     /* variables */  
19     int i, tid;  
20  
21     /* create parallel region, make both variable thread private */  
22     #pragma omp parallel private(i,tid)  
23     {  
24         /* get id of thread */  
25         tid = omp_get_thread_num();  
26  
27         /* divide loop iterations evenly by threads */  
28         #pragma omp for nowait  
29         for(i = 0; i < NUMITER; i++)  
30             printf("Thread: %d\riter=%d\n", tid, i); fflush(stdout);  
31  
32         /* one of these messages per thread */  
33         printf("Thread %d, almost...\n", tid); fflush(stdout);  
34         printf("Thread %d, done!\n", tid); fflush(stdout);  
35     }  
36  
37     /* only master thread prints this one */  
38     printf("All threads have finished!\n");  
39 }  
40
```

figura b 1 - Cláusula "nowait" adicionada à directiva for na linha 9.



```
willfredy@DESKTOP-CBRD131: /mnt/c/Users/willf/Downloads/drive-download-20250405T142442Z-001$ ./omp1  
Thread: 0   iter=0  
Thread: 0   iter=1  
Thread: 0   iter=2  
Thread: 0   iter=3  
Thread: 0   iter=4  
Thread: 0   iter=5  
Thread: 0   iter=6  
Thread: 0   iter=7  
Thread: 2   iter=16  
Thread: 2   iter=17  
Thread: 2   iter=18  
Thread: 2   iter=19  
Thread: 2   iter=20  
Thread: 2   iter=21  
Thread: 2   iter=22  
Thread: 1   iter=8  
Thread: 1   iter=9  
Thread: 1   iter=10  
Thread: 1   iter=11  
Thread: 1   iter=12  
Thread: 1   iter=13  
Thread: 1   iter=14  
Thread: 1   iter=15  
Thread: 3   iter=23  
Thread: 3   iter=24  
Thread: 3   iter=25  
Thread: 3   iter=26  
Thread: 3   iter=27  
Thread: 3   iter=28  
Thread: 3   iter=29  
Thread 2, almost..  
Thread 2, done!  
Thread 0, almost..  
Thread 0, done!  
Thread 1, almost..  
Thread 1, done!  
Thread 3, almost..  
Thread 3, done!  
All threads have finished!  
willfredy@DESKTOP-CBRD131: /mnt/c/Users/willf/Downloads/drive-download-20250405T142442Z-001$
```

figura b 2 - Execução do programa com a cláusula "nowait".

- **Resposta:** Ao utilizar a cláusula **nowait**, retiramos a barreira implícita que normalmente se encontra ao fim do loop **for**. Isto significa que cada thread pode avançar para as instruções seguintes sem ter de esperar que as outras threads concluam todas as iterações do loop.

No código, imediatamente após o loop **for**, cada thread executa os comandos:

```
printf("Thread %d, almost..\n", tid);
```

```
printf("Thread %d, done!\n", tid);
```

Sem a barreira, estas mensagens podem aparecer numa ordem diferente, consoante qual thread termina o loop mais rapidamente.

Resumindo:

- **Sem nowait (com barreira):** Todas as threads esperam o fim do loop **for** antes de imprimir as mensagens, resultando numa saída um pouco mais organizada (embora a ordem entre threads ainda possa variar).
- **Com nowait:** As threads começam a executar as linhas seguintes assim que terminam o seu trabalho no loop, o que pode fazer com que as mensagens apareçam de forma intercalada e numa ordem menos previsível.

Assim, a saída pode alterar-se em termos de ordem das mensagens, mas a funcionalidade do programa permanece a mesma.

c) Adicione uma barreira (`#pragma omp barrier`) entre as linhas 13 e 14. Compare a saída com a) e b).

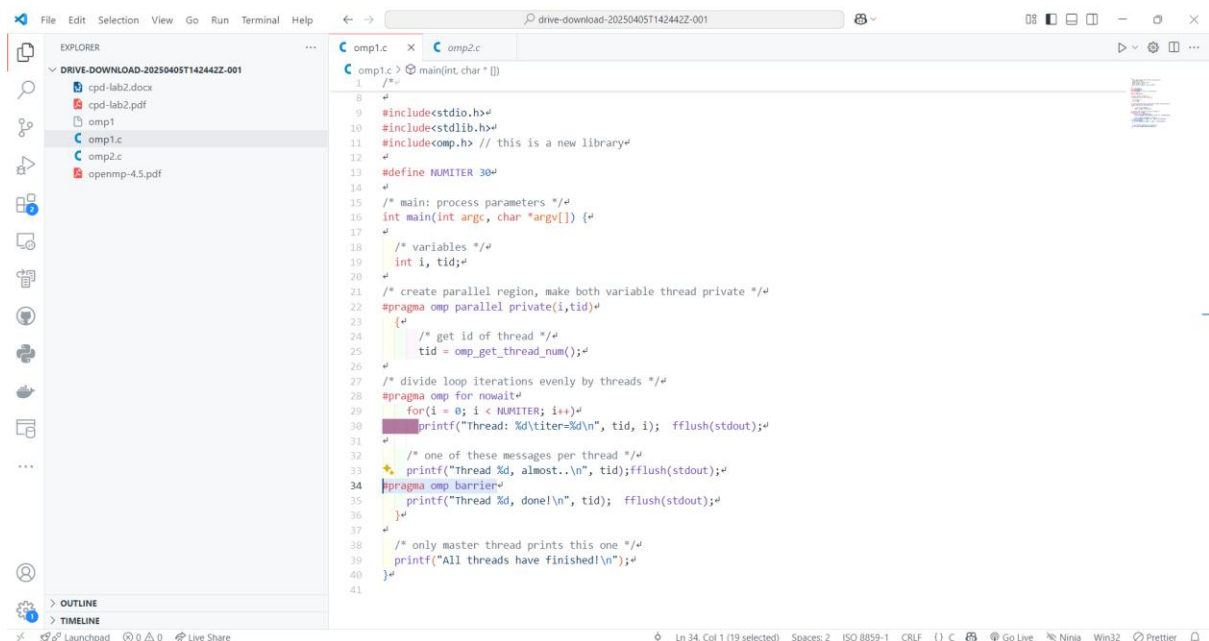


figura c 1 - Barreira `#pragma omp barrier` adicionada entre as linhas 33 e 34.

```
willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-202504051424422-001$ ./omp1
Thread: 0 iter=0
Thread: 0 iter=1
Thread: 0 iter=2
Thread: 0 iter=3
Thread: 0 iter=4
Thread: 0 iter=5
Thread: 0 iter=6
Thread: 0 iter=7
Thread: 0, almost...
Thread: 3 iter=23
Thread: 3 iter=24
Thread: 3 iter=25
Thread: 3 iter=26
Thread: 3 iter=27
Thread: 3 iter=28
Thread: 3 iter=29
Thread: 3, almost...
Thread: 1 iter=8
Thread: 1 iter=9
Thread: 1 iter=10
Thread: 1 iter=11
Thread: 1 iter=12
Thread: 1 iter=13
Thread: 1 iter=14
Thread: 1 iter=15
Thread: 1, almost...
Thread: 2 iter=16
Thread: 2 iter=17
Thread: 2 iter=18
Thread: 2 iter=19
Thread: 2 iter=20
Thread: 2 iter=21
Thread: 2 iter=22
Thread: 2, almost...
Thread: 1, done!
Thread: 2, done!
Thread: 3, done!
Thread: 0, done!
All threads have finished!
willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-202504051424422-001$
```

figura c 2 - Programa executado com a barreira `#pragma omp barrier`.

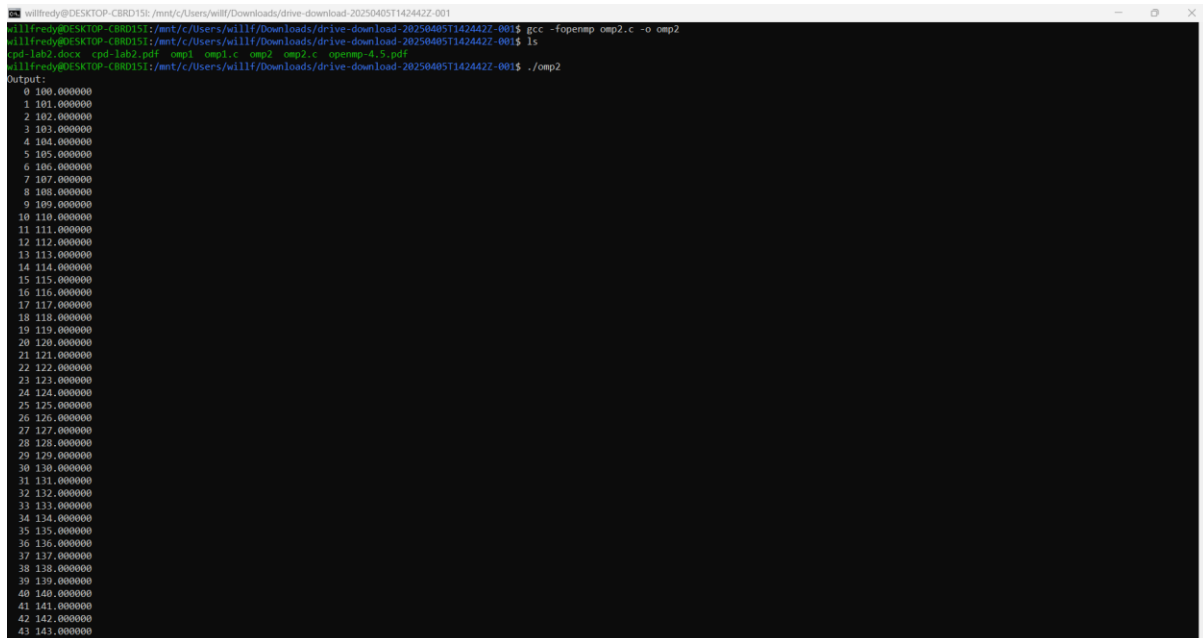
➤ **Comparação com a a linha a) e com a alínea b):**

- **Alínea a) – Sem modificações:** As threads executam o loop **for** e, ao fim do mesmo, existe uma barreira implícita. Assim, todas as threads esperam que o loop seja concluído antes de passarem para as impressões seguintes. A saída tende a ser mais organizada, pois as mensagens das threads são apresentadas apenas depois de todas terem terminado o loop.
- **Alínea b) – Com cláusula `nowait`:** Ao acrescentar a cláusula **`nowait`** à diretiva **`for`**, retiramos a barreira implícita. Cada thread pode avançar para os comandos seguintes assim que terminar o seu conjunto de iterações, sem aguardar as restantes. Isto faz com que as mensagens possam ser impressas de forma intercalada, resultando numa saída menos ordenada.
- **Alínea c) – Com barreira explícita:** Ao inserir a barreira com **`#pragma omp barrier`**, obrigamos todas as threads a esperarem até que todas concluem o loop **`for`** antes de prosseguir para as impressões. O efeito é semelhante ao da alínea **a)**, ou seja, uma saída onde as mensagens de “almost...” e “done!” aparecem de forma sincronizada entre as threads.

Em resumo, a adição da barreira explícita na alínea **c)** restabelece a sincronização entre as threads que foi perdida ao usar o **nowait** na alínea **b)**. Dessa forma, a ordem de execução torna-se mais previsível e organizada.

## Problema 2

a) Compile e execute a versão serial do código (omp2.c).



```
wilfredy@DESKTOP-CBRD15L:/mnt/c/Users/wilfredy/Downloads/drive-download-20250405T142442Z-001$ gcc -fopenmp omp2.c -o omp2
wilfredy@DESKTOP-CBRD15L:/mnt/c/Users/wilfredy/Downloads/drive-download-20250405T142442Z-001$ ls
cpd-1ab2.docx  cpd-1ab2.pdf  omp1  omp1.c  omp2  omp2.c  openmp-4.5.pdf
wilfredy@DESKTOP-CBRD15L:/mnt/c/Users/wilfredy/Downloads/drive-download-20250405T142442Z-001$ ./omp2
Output:
0 100.000000
1 101.000000
2 102.000000
3 103.000000
4 104.000000
5 105.000000
6 106.000000
7 107.000000
8 108.000000
9 109.000000
10 110.000000
11 111.000000
12 112.000000
13 113.000000
14 114.000000
15 115.000000
16 116.000000
17 117.000000
18 118.000000
19 119.000000
20 120.000000
21 121.000000
22 122.000000
23 123.000000
24 124.000000
25 125.000000
26 126.000000
27 127.000000
28 128.000000
29 129.000000
30 130.000000
31 131.000000
32 132.000000
33 133.000000
34 134.000000
35 135.000000
36 136.000000
37 137.000000
38 138.000000
39 139.000000
40 140.000000
41 141.000000
42 142.000000
43 143.000000
```

figura do segundo Problema a 1 - Versão serial (omp2.c) compilada.

**b)** Crie uma versão paralela simples (e incorreta) adicionando uma directiva paralela ao loop interno. Tente encontrar uma execução com uma saída diferente da versão serial. **Porquê isso pode acontecer?**

```
omp2.c
/* PALIN: PROBLEMA PARALELO */
29 int main(int argc, char *argv[]) {
30
31     /* VARIABLES */
32     int i, iter;
33
34     /* DECLARE VECTOR AND AUX DATA STRUCTURES */
35     double *V = (double *) malloc(TOTALSIZE * sizeof(double));
36
37     /* 1. INITIALIZE VECTOR */
38     for(i = 0; i < TOTALSIZE; i++) {
39         V[i] = 0.0 + i;
40     }
41
42     /* 2. ITERATIONS LOOP */
43     for(iter = 0; iter < NUMITER; iter++) {
44         #pragma omp parallel for
45         /* 2.1. PROCESS ELEMENTS */
46         for(i = 0; i < TOTALSIZE-1; i++) {
47             V[i] = f(V[i], V[i+1]);
48         }
49     }
50     /* 2.2. END ITERATIONS LOOP */
51
52     /* 3. OUTPUT FINAL VALUES */
53     printf("Output:\n");
54     for(i = 0; i < TOTALSIZE; i++) {
55         printf("%4d %f\n", i, V[i]);
56     }
57
58 }
59
60
```

figura do segundo problema b 1 - Código com a directiva paralela ao loop interno adicionada.

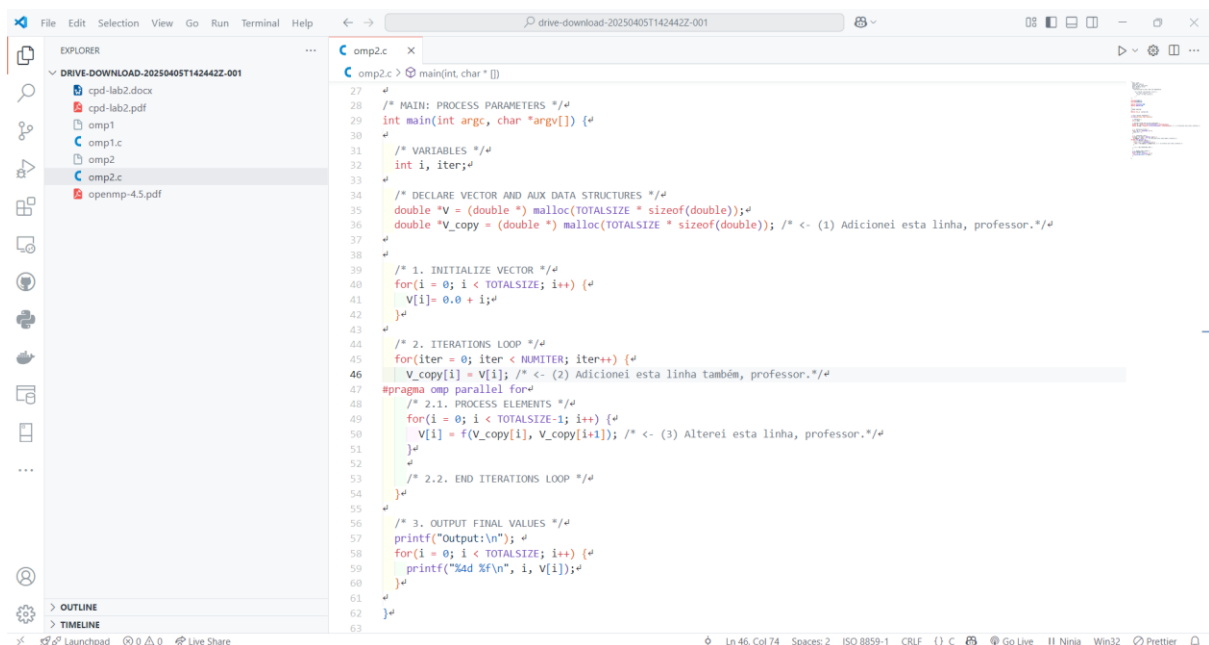
```
willfred@DESKTOP-CBRD151: /mnt/c/Users/willfred/Downloads/drive-download-20250405T142442Z-001$ gcc -fopenmp omp2.c -o omp2
willfred@DESKTOP-CBRD151: /mnt/c/Users/willfred/Downloads/drive-download-20250405T142442Z-001$ ./omp2
Output:
0 100.000000
1 101.000000
2 102.000000
3 103.000000
4 104.000000
5 105.000000
6 106.000000
7 107.000000
8 108.000000
9 109.000000
10 110.000000
11 111.000000
12 112.000000
13 113.000000
14 114.000000
15 115.000000
16 116.000000
17 117.000000
18 118.000000
19 119.000000
20 120.000000
21 121.000000
22 122.000000
23 123.000000
24 124.000000
25 125.000000
26 126.000000
27 127.000000
28 128.000000
29 129.000000
30 130.000000
31 131.000000
32 132.000000
33 133.000000
34 134.000000
35 135.000000
36 136.000000
37 137.000000
38 138.000000
39 139.000000
40 140.000000
41 141.000000
42 142.000000
43 143.000000
44 144.000000
45 145.000000
```

figura do segundo problema b 2 - Programa executado com a directiva paralela ao loop interno adicionada.

**Resposta:** Nesta versão, as iterações do loop interno são executadas em paralelo, mas a operação apresenta dependências entre elementos (por exemplo, o cálculo de  $V[i]$  depende de  $V[i+1]$ ). Na versão serial, o processamento ocorre de forma ordenada, garantindo que cada elemento seja atualizado de acordo com o valor mais recente dos elementos seguintes.

Ao paralelizar de forma simples, várias threads podem aceder e atualizar os elementos do vetor simultaneamente, causando condições de corrida (race conditions). Assim, a ordem de execução não fica garantida e o resultado final pode ser diferente, pois os valores lidos podem não ser os esperados se outra thread estiver a atualizar o mesmo elemento ao mesmo tempo.

c) Faça a correcção da versão paralela copiando o vector V da iteração i-1 antes de executar a iteração i e usando os valores copiados como argumentos para a função f na iteração i.



```
27  /*  
28  /* MAIN: PROCESS PARAMETERS */  
29  int main(int argc, char *argv[]) {  
30  
31  /* VARIABLES */  
32  int i, iter;  
33  
34  /* DECLARE VECTOR AND AUX DATA STRUCTURES */  
35  double *V = (double *) malloc(TOTALSIZE * sizeof(double));  
36  double *V_copy = (double *) malloc(TOTALSIZE * sizeof(double)); /* <- (1) Adicionei esta linha, professor.*/  
37  
38  
39  /* 1. INITIALIZE VECTOR */  
40  for(i = 0; i < TOTALSIZE; i++) {  
41      V[i] = 0.0 + i;  
42  }  
43  
44  /* 2. ITERATIONS LOOP */  
45  for(iter = 0; iter < NITER; iter++) {  
46      V_copy[i] = V[i]; /* <- (2) Adicionei esta linha também, professor.*/  
47      #pragma omp parallel for  
48      /* 2.1. PROCESS ELEMENTS */  
49      for(i = 0; i < TOTALSIZE-1; i++) {  
50          V[i] = f(V_copy[i], V_copy[i+1]); /* <- (3) Alterei esta linha, professor.*/  
51      }  
52      /* 2.2. END ITERATIONS LOOP */  
53  }  
54  
55  /* 3. OUTPUT FINAL VALUES */  
56  printf("Output:\n");  
57  for(i = 0; i < TOTALSIZE; i++) {  
58      printf("%4d %f\n", i, V[i]);  
59  }  
60  
61  }  
62  
63  }
```

figura do segundo problema c 1 - Código correcção da versão paralela feita com a cópia do vector V da iteração i-1.



```
willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-20250405T142442Z-001
willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-20250405T142442Z-001$ gcc -fopenmp omp2.c -o omp2
willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-20250405T142442Z-001$ ./omp2
Output:
0 0.000000
1 0.000000
2 0.000000
3 0.000000
4 0.000000
5 0.000000
6 0.000000
7 0.000000
8 0.000000
9 0.000000
10 0.000000
11 0.000000
12 0.000000
13 0.000000
14 0.000000
15 0.000000
16 0.000000
17 0.000000
18 0.000000
19 0.000000
20 0.000000
21 0.000000
22 0.000000
23 0.000000
24 0.000000
25 0.000000
26 0.000000
27 0.000000
28 0.000000
29 0.000000
30 0.000000
31 0.000000
32 0.000000
33 0.000000
34 0.000000
35 0.000000
36 0.000000
37 0.000000
38 0.000000
39 0.000000
40 0.000000
41 0.000000
42 0.000000
43 0.000000
44 0.000000
45 0.000000
```

figura do segundo problema c 2 - Programa executado com a correccção da versão paralela feita com a cópia do vector V da iteração i-1.

d) Escreva uma modificação na versão paralela que evite copiar o vector V em todas as iterações.

```
File Edit Selection View Go Run Terminal Help
drive-download-20250405T142442Z-001

EXPLORER
  DRIVE-DOWNLOAD-20250405T142442Z-001
    cpd-lab2.docx
    cpd-lab2.pdf
    omp1
    omp1.c
    omp2
    omp2.c
    openmp-4.5.pdf

omp2.c
29 int main(int argc, char *argv[]) {
30     /* DECLARE VECTOR AND AUX DATA STRUCTURES */
31     double *V = (double *) malloc(TOTALSIZE * sizeof(double));
32     double *V_copy = (double *) malloc(TOTALSIZE * sizeof(double)); /* <- (1) Adicionei esta linha, professor.*/
33     double *W = (double *) malloc(TOTALSIZE * sizeof(double));
34
35     /* 1. INITIALIZE VECTOR */
36     for(i = 0; i < TOTALSIZE; i++) {
37         V[i] = 0.0 + i;
38     }
39
40     /* 2. ITERATIONS LOOP */
41     for(iter = 0; iter < NUMITER; iter++) {
42
43         #pragma omp parallel for
44         /* 2.1. PROCESS ELEMENTS */
45         for(i = 0; i < TOTALSIZE-1; i++) {
46             W[i] = f(V[i], V[i+1]); /* <- (3) Alterei esta linha, professor.*/
47         }
48         W[TOTALSIZE - 1] = V[TOTALSIZE - 1]; // Linha adicionada: manter o último elemento
49
50         /* Troca dos ponteiros */
51         double *temp = V; // Linha adicionada
52         V = W; // Linha adicionada
53         W = temp;
54     }
55
56     /* 2.2. END ITERATIONS LOOP */
57
58     if (NUMITER % 2 == 1) {
59         double *temp = V; // Linha adicionada
60         V = W; // Linha adicionada
61         W = temp; // Linha adicionada
62     }
63 }
Ln 67, Col 2 Spaces: 2 ISO 8859-1 CRLF ( ) C Go Live II Ninja Win32 Prettier
```

figura do segundo problema d 1 - Código com a modificação da versão paralela que evita copiar o vector V em todas as iterações.

```
willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-202504051424422-001$ gcc -fopenmp omp2.c -o omp2
willfredy@DESKTOP-CBRD151: /mnt/c/Users/willf/Downloads/drive-download-202504051424422-001$ ./omp2
Output:
0 100.000000
1 101.000000
2 102.000000
3 103.000000
4 104.000000
5 105.000000
6 106.000000
7 107.000000
8 108.000000
9 109.000000
10 110.000000
11 111.000000
12 112.000000
13 113.000000
14 114.000000
15 115.000000
16 116.000000
17 117.000000
18 118.000000
19 119.000000
20 120.000000
21 121.000000
22 122.000000
23 123.000000
24 124.000000
25 125.000000
26 126.000000
27 127.000000
28 128.000000
29 129.000000
30 130.000000
31 131.000000
32 132.000000
33 133.000000
34 134.000000
35 135.000000
36 136.000000
37 137.000000
38 138.000000
39 139.000000
40 140.000000
41 141.000000
42 142.000000
43 143.000000
44 144.000000
45 145.000000
```

figura do segundo problema d 2 - Execução do programa com a modificação da versão paralela que evita copiar o vector V em todas as iterações.

### 3. Desafios

Neste trabalho, o principal desafio foi adaptar um código sequencial para uma versão paralela, mantendo a lógica e o resultado corretos. Para isso, foi necessário compreender bem como funciona a execução em paralelo com OpenMP, principalmente no que diz respeito às dependências entre iterações dos ciclos.

A estrutura do software ficou organizada da seguinte forma:

- **Inicialização:** O vetor principal é criado e preenchido com valores iniciais simples, para facilitar a verificação dos resultados.
- **Processamento:** A parte mais importante do programa está no ciclo de iterações, onde o vetor é atualizado várias vezes. Nesta fase, implementámos diferentes versões:
  - Uma versão **sequencial**, onde tudo é feito de forma linear.
  - Uma versão **paralela incorreta**, onde tentámos paralelizar diretamente o ciclo interno, o que gerou resultados inconsistentes devido à existência de dependências.
  - Uma versão **corrigida**, onde criámos uma cópia auxiliar do vetor em cada iteração para evitar as dependências, garantindo resultados corretos.

- Por fim, uma versão **otimizada**, onde evitámos fazer a cópia do vetor a cada iteração, usando dois vetores e alternando entre eles. Assim, melhorámos a eficiência sem comprometer a correção dos dados.

Durante o desenvolvimento, também foi necessário compreender bem como funcionam as diretivas **#pragma omp**, especialmente as cláusulas **parallel** for, **nowait** e **barrier**. Esses conceitos foram fundamentais para garantir uma execução paralela eficiente e correta.

## 4. Referências Bibliográficas

<http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>

## 5. Repositório GitHub

[https://github.com/Willfredy-Vieira-Dias/Laborat-rios\\_de\\_CPD](https://github.com/Willfredy-Vieira-Dias/Laborat-rios_de_CPD)

Coloque aqui o link do repositório e envie o convite de colaborador para o utilizador GitHub **joaojdacosta**.