

Universidade Federal do Rural do Semi-Árido
Centro Multidisciplinar de Pau dos Ferros
Departamento de Engenharias e Tecnologia
PEX1272 - Programação Concorrente e
Distribuída

Docente: Ítalo Assis

Discente: João Gustavo Souza Lima

Lista de Exercícios

Programação de memória compartilhada com OpenMP

Questões respondidas:

Questão 1 - Peso 1

Questão 2 - Peso 1

Questão 3 - Peso 1

Questão 4 - Peso 1

Questão 5 - Peso 1

Questão 9 - Peso 1

Questão 10 - Peso 1

Questão 13 - Peso 2

1. Baixe o arquivo `omp_trap_1.c` do site do livro e exclua a diretiva `critical`. Compile e execute o programa com cada vez mais *threads* e valores cada vez maiores de *n*.

(a) Quantas *threads* e quantos trapézios são necessários antes que o resultado esteja incorreto?

Thread	número de trapézios							
	16	160	1600	16000	160000	1600000	16000000	160000000
1	3.3398437500000e-01	3.3333984375000e-01	3.3333339843750e-01	3.3333333984376e-01	3.3333333333984e-01	3.3333333333339e-01	3.3333333333330e-01	3.3333333333336e-01
4	3.3398437500000e-01	1.0416992187500e-01	3.3333339843750e-01	3.3333333984375e-01	3.3333333333984e-01	3.3333333333339e-01	3.3333333333333e-01	3.3333333333333e-01
8	8.1787109375000e-02	3.3333984375000e-01	3.0924484863281e-01	3.3333333984375e-01	2.3437500004883e-01	3.3333333333340e-01	3.3333333333333e-01	3.3333333333331e-01
16	1.0119628906250e-01	1.0441284179687e-01	1.7358402099609e-01	2.3006184948730e-01	2.0345052083740e-01	2.0499674479171e-01	3.1567382812500e-01	3.3333333333333e-01

De acordo com as execuções para obter resultados corretos, deve ter menos que 4 threads e o número de trapézios 160, a partir desses dois valores a chance do resultado ser errado devido à condição de corrida.

(b) Como o aumento do número de trapézios influencia nas chances do resultado ser incorreto?

Quando aumentamos o número de trapézios, cada thread executa muito mais iterações no loop. Isso aumenta significativamente o número de acessos à variável compartilhada usada para somar os resultados parciais. Com mais acessos concorrentes, cresce a chance de duas threads tentarem atualizar essa variável ao mesmo tempo, o que torna muito mais provável a ocorrência de condição de corrida.

(c) Como o aumento do número de *threads* influencia nas chances do resultado ser incorreto?

O aumento do número de threads aumenta as chances de resultado incorreto porque mais threads tentam atualizar a variável global ao mesmo tempo, elevando a probabilidade de colisão na operação `global_result += my_result` e, portanto, de condição de corrida que geram valores errados.

script utilizado para execução:

```
#!/bin/bash
#SBATCH --nodes=1
#Número de tarefas por Nó
#SBATCH --ntasks-per-node=1
#Número total de tarefas MPI
#SBATCH --ntasks=1
#Número de threads
#SBATCH --cpus-per-task=16
#Fila (partition) a ser utilizada
#SBATCH -p sequana_cpu_dev
#Nome do job
#SBATCH -J trap_openmp
#Utilização exclusiva dos nós durante a execução do job
#SBATCH --exclusive
#Arquivo de saída
#SBATCH --output=saida_trap_%j.log

# Exibe os nós alocados para o job
echo "Nós alocados: $SLURM_JOB_NODELIST"

# Caminho do diretório de execução
cd /scratch/pex1272-ufersa/joao.lima2/

# Carregar compilador
module load gcc/14.2.0_sequana

gcc -fopenmp -o omp_trap1 omp_trap1.c

# Executar com diferentes números de threads e trapézios
for threads in 1 4 8 16; do
    export OMP_NUM_THREADS=$threads

    echo "Executando com $threads threads"

    for n in 16 160 1600 16000 160000 1600000 16000000 160000000; do
        echo "Número de trapézios: $n"
        echo "0 1 $n" | ./omp_trap1 $threads
        echo "-----"
    done
done
```

2. Baixe o arquivo `omp_trap1.c` do site do livro. Modifique o código para que

- ele use o primeiro bloco de código da página 222 do livro e
- o tempo usado pelo bloco paralelo seja cronometrado usando a função OpenMP `omp_get_wtime()`. A sintaxe é

```
double omp_get_wtime(void)
```

Ele retorna o número de segundos que se passaram desde algum tempo no passado. Para obter detalhes sobre cronometragem, consulte a Seção 2.6.4. Lembre-se também de que o OpenMP possui uma diretiva de barreira:

```
# pragma omp barrier
```

Agora encontre um sistema com pelo menos dois núcleos e cronometre o programa com

- uma *thread* e um grande valor de n , e
- duas *threads* e o mesmo valor de n .

(a) O que acontece?

```
Enter a, b, and n
10
20
1000000000000000

With n = 276447232 trapezoids, our estimate
of the integral from 10.000000 to 20.000000 = 2.33333333333287e+03
Elapsed time = 0.764024 seconds
→ codigos modificados ./omp_trap1 2
Enter a, b, and n
10
20
1000000000000000

With n = 276447232 trapezoids, our estimate
of the integral from 10.000000 to 20.000000 = 2.33333333333361e+03
Elapsed time = 0.771114 seconds
```

Com apenas uma thread, o programa roda sequencialmente, como esperado, e o tempo de execução foi de aproximadamente 0,76 segundos.

Com duas threads, a segunda thread precisa aguardar a liberação da região crítica pela primeira para poder realizar sua soma parcial. Assim, o tempo de execução não melhora e pode até piorar levemente, devido à um overhead adicional de gerenciamento das threads.

(b) Baixe o arquivo `omp_trap2b.c` do site do livro. Como seu desempenho se compara?

```
Enter a, b, and n
10
20
10000000000000000
With n = 276447232 trapezoids, our estimate
of the integral from 10.000000 to 20.000000 = 2.33333333333287e+03 Elapsed time
= 8.023196e-01 seconds
→ ch5 ./omp_trap2b 2
Enter a, b, and n
10
20
10000000000000000
With n = 276447232 trapezoids, our estimate
of the integral from 10.000000 to 20.000000 = 2.33333333333361e+03 Elapsed time
= 4.010610e-01 seconds
```

Agora as threads realmente trabalham em paralelo.

Nos testes realizados, o tempo com uma thread foi de aproximadamente 0,80 segundos, enquanto com duas threads caiu para cerca de 0,40 segundos.

Isso mostra uma redução de tempo praticamente pela metade, indicando uma paralelização eficiente. Já no `trap1` com a modificação pedida há a serialização do código que executa a mesma função sendo maior o tempo com o aumento do número de threads.

3. Suponha que no incrível computador Bleeblon, variáveis com tipo float possam armazenar três dígitos decimais. Suponha também que os registradores de ponto flutuante do Bleeblon possam armazenar quatro dígitos decimais e que, após qualquer operação de ponto flutuante, o resultado seja arredondado para três dígitos decimais antes de ser armazenado. Agora suponha que um programa C declare um *array* *a* da seguinte forma:

```
float a[] = {4.0, 3.0, 3.0, 1000.0};
```

(a) Qual é a saída do seguinte bloco de código se ele for executado no Bleeblon? Justifique sua resposta.

```
int i ;  
float sum = 0.0;  
for (i = 0; i < 4; i++)  
    sum += a[i];  
printf ("sum = %4.1f\n", sum );
```

A saída será 1010.0

Operação	Iterações			
	0	1	2	3
fetch	$0.00 \times 10^0 \mid 4.00 \times 10^0$	$4.00 \times 10^0 \mid 3.00 \times 10^0$	$7.00 \times 10^0 \mid 3.00 \times 10^0$	$1.00 \times 10^1 \mid 1.00 \times 10^3$
compare exponents	$0.00 \times 10^0 \mid 4.00 \times 10^0$	$4.00 \times 10^0 \mid 3.00 \times 10^0$	$7.00 \times 10^0 \mid 3.00 \times 10^0$	$1.00 \times 10^1 \mid 1.00 \times 10^3$
shift	$0.00 \times 10^0 \mid 4.00 \times 10^0$	$4.00 \times 10^0 \mid 3.00 \times 10^0$	7.00×10^0	$0.01 \times 10^3 \mid 1.00 \times 10^3$
add	4.00×10^0	7.00×10^0	10.00×10^0	1.01×10^3
normalize	4.00×10^0	7.00×10^0	1.00×10^1	1.01×10^3
round	4.00×10^0	7.00×10^0	1.00×10^1	1.01×10^3
store	4.00×10^0	7.00×10^0	1.00×10^1	1.01×10^3

Após cada operação de soma, o resultado é arredondado para três dígitos decimais antes de ser armazenado.
Como todas as operações intermediárias resultam em valores que podem ser representados exatamente com três dígitos significativos, não há erro de arredondamento perceptível, e o valor final armazenado.

(b) Agora considere o seguinte código:

```
int i;  
float sum = 0.0;  
#pragma omp parallel for num threads (2) reduction (+:sum)  
for (i=0;i<4;i++)  
    sum += a[i];  
printf("sum = %4.1f\n", sum );
```

Suponha que o sistema operacional atribua as iterações $i = 0, 1$ à *thread* 0 e $i = 2, 3$ à *thread* 1. Qual é a saída deste código no Bleeblon? Justifique sua resposta.

Thread 0: valor final: 7

Operação	Iterações	
	0	1
fetch	$0.00 \times 10^0 \mid 4.00 \times 10^0$	$4.00 \times 10^0 \mid 3.00 \times 10^0$
compare	$0.00 \times 10^0 \mid 4.00 \times 10^0$	$4.00 \times 10^0 \mid 3.00 \times 10^0$
shift	$0.00 \times 10^0 \mid 4.00 \times 10^0$	$4.00 \times 10^0 \mid 3.00 \times 10^0$
add	4.00×10^0	7.00×10^0
normalize	4.00×10^0	7.00×10^0
round	4.00×10^0	7.00×10^0
store	4.00×10^0	7.00×10^0

Thread 1: valor final = 1000

Operação	Iterações	
	2	3
fetch	$0.00 \times 10^0 \mid 3.00 \times 10^0$	$3.00 \times 10^0 \mid 1.00 \times 10^3$
compare	$0.00 \times 10^0 \mid 3.00 \times 10^0$	$1.00 \times 10^1 \mid 1.00 \times 10^3$
shift	$0.00 \times 10^0 \mid 3.00 \times 10^0$	$0.003 \times 10^3 \mid 1.00 \times 10^3$
add	3.00×10^0	1.003×10^3
normalize	3.00×10^0	1.003×10^3
round	3.00×10^0	1.00×10^3
store	3.00×10^0	1.00×10^3

(Resultado errado pois ele só armazena 3 dígitos decimais e o número correto precisaria de 4.)

Reduction: Valor final 1000

Operação	Reduction
fetch	$7.00 \times 10^0 \mid 1.00 \times 10^3$
compare	$7.00 \times 10^0 \mid 1.00 \times 10^3$
shift	$0.007 \times 10^3 \mid 1.00 \times 10^3$
add	1.007×10^3
normalize	1.007×10^3
round	1.00×10^3
store	1.00×10^3

(resultado errado novamente, pelo mesmo motivo)

4. Escreva um programa OpenMP que determine o escalonamento padrão de laços for paralelos. Sua entrada deve ser o número de iterações e quantidade de *threads* e sua saída deve ser quais iterações de um laço for paralelizado são executadas por qual *thread*. Por exemplo, se houver duas *threads* e quatro iterações, a saída poderá ser:

Thread 0: Iterações 0 -- 1

Thread 1: Iterações 2 -- 3

- (a) De acordo com a execução do seu programa, qual é o escalonamento padrão de laços for paralelos de um programa OpenMP? Porque?

Código:

```
int main(int argc, char* argv[]) {
    if (argc != 3) {
        printf("Uso: %s <num_iteracoes> <num_threads>\n", argv[0]);
        return 1;
    }
    int n = atoi(argv[1]);
    int thread_count = atoi(argv[2]);

    omp_set_num_threads(thread_count);

    int first[thread_count];
    int last[thread_count];

    for (int i = 0; i < thread_count; i++) {
        first[i] = -1;
        last[i] = -1;
    }

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        int my_rank = omp_get_thread_num();

        if (first[my_rank] == -1) first[my_rank] = i;
        last[my_rank] = i;
    }

    for (int t = 0; t < thread_count; t++) {
        if (first[t] != -1)
            printf("Thread %d: Iterações %d -- %d\n", t, first[t], last[t]);
        else
            printf("Thread %d: não executou iterações.\n", t);
    }

    return 0;}
```

O escalonamento padrão é o deslocamento em blocos, ou seja, static.

Com base no log de execução, observei um padrão que mostra que o laço foi dividido em blocos contíguos iguais, cada thread pega um bloco fixo exatamente como o *schedule(static)* faz.

```
Nós alocados: sdumont6165
Executando com 8 threads
Iterações = 16
Thread 0: Iterações 0 -- 1
Thread 1: Iterações 2 -- 3
Thread 2: Iterações 4 -- 5
Thread 3: Iterações 6 -- 7
Thread 4: Iterações 8 -- 9
Thread 5: Iterações 10 -- 11
Thread 6: Iterações 12 -- 13
Thread 7: Iterações 14 -- 15
```

script utilizado para execução:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=16
#SBATCH -p sequana_cpu_dev
#SBATCH -J omp_schedule
#SBATCH --exclusive
#SBATCH --output=saida_schedule_%j.log

echo "Nós alocados: $SLURM_JOB_NODELIST"

cd /scratch/pex1272-ufersa/joao.lima2/

module load gcc/14.2.0_sequana

# Compilar seu código
gcc -fopenmp -o questao4 questao4.c

# Executar com diferentes quantidades de threads
for threads in 8 16; do
    export OMP_NUM_THREADS=$threads

    echo "Executando com $threads threads"

    for n in 16 40 100 1000; do
        echo "Iterações = $n"
        ./questao4 $n $threads
        echo "-----"
    done
done
```

5. Considere o seguinte laço:

```
a[0] = 0;
```

```
for ( i = 1; i < n ; i++)
```

```
  a[i] = a[i-1] + i;
```

Há claramente uma dependência no laço já que o valor de $a[i]$ não pode ser calculado sem o valor de $a[i-1]$. Sugira uma maneira de eliminar essa dependência e paralelizar o laço.

O laço está calculando a soma dos n primeiros inteiros, pode ser expressa pela fórmula $((i * (i + 1)) / 2)$

solução:

```
#pragma omp parallel for num_threads(thread_count) \
default(none) private(i) shared(a, n)
  for (i = 0; i < n; i++) {
    a[i] = (i * (i + 1)) / 2;
```

9. Lembre-se do exemplo de multiplicação de matrizes e vetores com a entrada 8000×8000 . Assuma que uma linha de *cache* contém 64 *bytes* ou 8 doubles.

M[8000 linhas x 8000 colunas]

x[8000 linhas]

y[8000 linhas]

4 threads [0,1,2,3], no caso serão atribuídos $8000/4=2000$ linhas para cada thread.

thread 0 = y[0], ... , y[1999]

thread 1 = y[2000], ... , y[3999]

thread 2 = y[4000], ... , y[5999]

thread 3 = y[6000] , ... , y[7999]

(a) Suponha que a *thread* 0 e a *thread* 2 sejam atribuídas a processadores diferentes. É possível que ocorra um falso compartilhamento entre as *threads* 0 e 2 para alguma parte do vetor *y*? Por que?

não há possibilidade de ocorrer o falso compartilhamento, pois essa faixa da thread 0 não acessa espaço da memória utilizada pela thread 2. os blocos que cada thread atualiza são bem separados: thread 0 atualiza índices em torno de 0–1999 e thread 2 em 4000–5999. A menor distância entre os elementos que elas atualizam é muito maior que o tamanho de uma linha de cache (8 elementos).

- (b) E se a *thread* 0 e a *thread* 3 forem atribuídas a processadores diferentes? É possível que ocorra um falso compartilhamento entre elas para alguma parte de *y*?

Não, pois como o último acesso da *thread* 0 poderá ser é [1999,2006] e o menor endereço buscado pela *thread* 3 é [6000], ou seja, não há risco de falso compartilhamento, nesse caso, pois a faixa não acessa endereço pertencente a *thread* 3

10. Embora `strtok_r` seja *thread-safe*, ele tem a propriedade bastante infeliz de modificar a *string* de entrada. Escreva um método para gerar *tokens* que seja *thread-safe* e não modifique a *string* de entrada.

```
char *tokenize(const char *text, const char *delim, char **my_line){
    int start = 0, end = 0;

    if (text != NULL) {
        *my_line = malloc(strlen(text) + 1);
        if (!*my_line) {
            exit(1);
        }
        strcpy(*my_line, text);
    }

    int str_len = strlen(*my_line);
    while (start < str_len && strchr(delim, (*my_line)[start])) {
        start++;
    }
    if (start == str_len) {
        *my_line = NULL;
        return NULL;
    }

    for (end = start; end < str_len; end++) {
        if (strchr(delim, (*my_line)[end])) {
            break;
        }
    }
    int tok_len = end - start;
    char *tok = malloc(tok_len + 1);
    if (!tok) {
        exit(1);
    }
    strncpy(tok, *my_line + start, tok_len);
    tok[tok_len] = '\0';

    *my_line += end;
    return tok;
}
```

Saida:

```
oi, tudo bem?  
queria muito passar nessa disciplina  
ai eu me formaria semestre que vem  
estou dando meu maximo  
Thread 0 terminou a linha -> oi, tudo bem?  
e os tokens foram -> oi -- tudo -- bem?  
  
Thread 1 terminou a linha -> queria muito passar nessa disciplina  
e os tokens foram -> queria -- muito -- passar -- nessa -- disciplina  
  
Thread 3 terminou a linha -> estou dando meu maximo  
e os tokens foram -> estou -- dando -- meu -- maximo  
  
Thread 2 terminou a linha -> ai eu me formaria semestre que vem  
e os tokens foram -> ai -- eu -- me -- formaria -- semestre -- que -- vem
```

13. Count sort é um algoritmo de ordenação serial simples que pode ser implementado da seguinte forma:

```
void Count_sort(int a[], int n) {
    int i, j, count;
    int* temp = malloc(n*sizeof(int));
    for (i = 0; i < n; i++) {
        count = 0;
        for (j = 0; j < n; j++)
            if (a[j] < a[i])
                count++;
            else if (a[j] == a[i] && j < i)
                count++;
        temp[count] = a[i];
    }
    memcpy(a, temp, n*sizeof(int));
    free(temp);
}
```

A ideia básica é que para cada elemento $a[i]$ na lista a , contemos o número de elementos da lista que são menores que $a[i]$. Em seguida, inserimos $a[i]$ em uma lista temporária usando o índice determinado pela contagem. Há um pequeno problema com esta abordagem quando a lista contém elementos iguais, uma vez que eles podem ser atribuídos ao mesmo slot na lista temporária. O código lida com isso incrementando a contagem de elementos iguais com base nos índices. Se $a[i] == a[j]$ e $j < i$, então contamos $a[j]$ como sendo "menor que" $a[i]$.

Após a conclusão do algoritmo, sobrescrevemos o array original pelo array temporário usando a função da biblioteca de strings `memcpy`.

(a) Se tentarmos paralelizar o laço `for i` (o laço externo), quais variáveis devem ser privadas e quais devem ser compartilhadas?

`Private(i,j,count) , Shared(temp,a,n)`

(b) Se paralelizarmos o laço `for i` usando o escopo especificado na parte anterior, haverá alguma dependência de dados no laço? Explique sua resposta.

`Não há dependência de dados entre as iterações do laço externo.`

`Cada iteração calcula sua própria contagem usando apenas dados lidos de $a[]$, que é compartilhado apenas para leitura. As escritas em $temp[]$ nunca colidem, pois cada $count$ é único. Portanto, as iterações são independentes e podem ser executadas em paralelo sem riscos de condição de corrida.`

(c) Podemos paralelizar a chamada para memcpy? Podemos modificar o código para que esta parte da função seja paralelizável?

Não podemos paralelizar diretamente o memcpy, mas podemos modificar o código, basta substituir o memcpy por um for e paralelizar esse laço:

```
#pragma omp parallel for
for (int k = 0; k < n; k++)
    a[k] = temp[k];
```

(d) Escreva um programa em C que inclua uma implementação paralela do Count sort.

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

void count_sort(int a[], int n);
int main() {
    int a[26] = {10,0,1,5,2,1,0,4,12,4,5,6,7,3,1,2,6,4,33,12,0,12,3,4,5,0};
    int n = 26;
    printf("Array antes:\n");
    for(int i=0; i<n; i++)
        printf("%d ", a[i]);
    count_sort(a, n);

    printf("\n\nArray depois:\n");
    for(int i=0; i<n; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}

void count_sort(int a[], int n) {
    int* temp = malloc(n * sizeof(int));
    int i, j, count;
    #pragma omp parallel shared(a, temp, n) private(i, j, count){

        #pragma omp parallel for
        for (i = 0; i < n; i++) {
            count = 0;
            for (j = 0; j < n; j++) {
                if (a[j] < a[i])
                    count++;
                else if (a[j] == a[i] && j < i)
                    count++;
            }
            temp[count] = a[i];
        }

        #pragma omp parallel for
        for (int k = 0; k < n; k++) {
            a[k] = temp[k];
        }
    }
    free(temp);
}
```

saída:

```
Nós alocados: sdumont6165
Executando Count Sort com 8 threads
Array antes:
10 0 1 5 2 1 0 4 12 4 5 6 7 3 1 2 6 4 33 12 0 12 3 4 5 0

Array depois:
0 0 0 0 1 1 1 2 2 3 3 4 4 4 4 5 5 5 6 6 7 10 12 12 12 33
```

(e) Como o desempenho da sua paralelização do Count sort se compara à classificação serial? Como ela se compara à função serial qsort?

```
int main() {
    int n;
    printf("Digite o tamanho do array: ");
    scanf("%d", &n);

    int* a = malloc(n * sizeof(int));
    int* b = malloc(n * sizeof(int));
    int* c = malloc(n * sizeof(int));

    arrayValor(a, n);
    memcpy(b, a, n * sizeof(int));
    memcpy(c, a, n * sizeof(int));
    double inicio, final;

    // Count Sort Paralelo
    inicio = omp_get_wtime();
    count_sort_parallel(a, n);
    final = omp_get_wtime();
    printf("\nTempo Count Sort Paralelo: %f s\n", final - inicio);

    // Count Sort Serial
    inicio = omp_get_wtime();
    count_sort_serial(b, n);
    final = omp_get_wtime();
    printf("Tempo Count Sort Serial: %f s\n", final - inicio);
    // qsort padrão
    inicio = omp_get_wtime();
    qsort(c, n, sizeof(int), comp);
    final = omp_get_wtime();
    printf("Tempo qsort: %f s\n", final - inicio);

    free(a);
    free(b);
    free(c);

    return 0;}

```

saída:

```
Nós alocados: sdumont6165
Executando comparações com 8 threads
Digite o tamanho do array:
Tempo Count Sort Paralelo: 0.001507 s
Tempo Count Sort Serial: 0.001142 s
Tempo qsort: 0.000118 s
```

O Count Sort paralelizado não supera a versão serial; pelo contrário, ele fica mais lento devido ao alto custo computacional do algoritmo e ao overhead das threads.

Quando comparado ao qsort, a diferença é ainda maior: mesmo totalmente serial, o qsort é muito mais rápido, pois usa algoritmos com complexidade muito menor.

script usado para executar a d):

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH -p sequana_cpu_dev
#SBATCH -J count_sort
#SBATCH --exclusive
#SBATCH --output=saida_countsort_%j.log

# Exibe os nós alocados
echo "Nós alocados: $SLURM_JOB_NODELIST"

# Diretório de execução
cd /scratch/pex1272-ufersa/joao.lima2/codigosModificados/count_sort_paralelo/

# Carregar compilador
module load gcc/14.2.0_sequana

# Compilar programa
gcc -fopenmp -o count_sort count_sort.c

# Definir 8 threads
export OMP_NUM_THREADS=8

echo "Executando Count Sort com 8 threads"
./count_sort
```

script usado para executar a e):

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH -p sequana_cpu_dev
#SBATCH -J count_compare
#SBATCH --exclusive
#SBATCH --output=saida_countcompare_%j.log

echo "Nós alocados: $SLURM_JOB_NODELIST"

cd /scratch/pex1272-ufersa/joao.lima2/codigosModificados/count_sort_paralelo/

module load gcc/14.2.0_sequana

gcc -fopenmp -O3 -o count_compare count_compare.c

export OMP_NUM_THREADS=8

echo "Executando comparações com 8 threads"
echo 1600 | ./count_compare
```

Referência

PACHECO, Peter S. An introduction to parallel programming. Amsterdam Boston: Morgan Kaufmann, c2011. xix, 370 p. ISBN: 9780123742605

