**Guião 5**

**Resumo introdutório**

**Número de *threads* usadas pelo *openMP*:**

* Por omissão, = ao número de cores virtuais;
* Controlada pela flag: *--cpus-per-tasks* na linha de compilação;
* Controlada pela linha de comandos da bash: *export OMP\_NUM\_THREADS=N*
* Codificado na aplicação através de instruções omp: #pragma omp parallel num\_threads(2)

**Compilar para OpenMP:**

* Carregar módulo gcc que suporta openMP: *module load gcc/7.2.0*
* Compilar programa com suporte openMP: *gcc -O2 -fopenmp -lm sample.c*
* Executar programa no cluster com especificação de número de threads:
  + *srun --partition=cpar --cpus-per-task=2 ./a.out*
* Executar o comando para obter estatísticas de execução:
  + *srun --partition=cpar --cpus-per-task=2 perf stat ./a.out*

**Diretivas OMP**

* #pragma omp for: distribui um conjunto de iterações por um conjunto de threads.
* #pragma omp master: identifica uma seção de código que apenas deve ser executada pela thread mestre.
* #pragma omp single: identifica uma seção de código que apenas deve ser executada por uma única thread disponível.
* #pragma omp critical: identifica uma seção de código que apenas deve ser executada ao mesmo tempo por apenas uma thread.
* #pragma omp barrier: identifica o ponto em que todas as threads devem esperar pelas outras;
* #pragma omp ordered: identifica um bloco de código que deve ser executado em ordem sequencial, isto é, uma thread depois da outra ter terminado a sua execução.

**Diretivas Schedule** ([*https://610yilingliu.github.io/2020/07/15/ScheduleinOpenMP*](https://610yilingliu.github.io/2020/07/15/ScheduleinOpenMP)), servem para destribuir a iterações pelas diferentes threads em loops for. SINTAXE: #pragma omp parallel for schedule(scheduling-type)

* **shedule static**:
  + **sintaxe**: #pragma omp parallel for schedule(static,chunk-size)
  + **chunck-size**: quantidade de grupos de iterações a dividir para atribuir às threads existentes **pela ordem** é o que distingue este método dos outros.
  + If you specify chunk-size variable, the iterations will be divide into ***iter\_size / chunk\_size chunks***.



* + But what if iter\_size / chunk\_size is larger than the number of threads in your computer, or number of threads you specified in omp\_set\_num\_threads(thread\_num)?

**R**: OpenMP will still split task into 7 chunks, but distributes the chunks to threads in a circular order, like the following figure shows:



* + estes dois formatos são equivalentes
    - #pragma omp parallel for e
    - #pragma omp parallel for schedule(static,1)
* **shedule dinamic**:
  + **sintaxe**: #pragma omp parallel for schedule(dynamic,chunk-size).
  + Continua a dividir as tarefas em *iter\_size / chunk\_size chunks*, mas agora a atribuição às threads não segue nenhum ordem em especifico.
* **Comparing with static Method**:
  + **Pros**: The dynamic scheduling type is appropriate when the iterations require different computational costs. This means that the iterations are not as balance as static method between each other.
  + **Cons**: The dynamic scheduling type has higher overhead then the static scheduling type because it dynamically distributes the iterations during the runtime.
* **schedule sguided:**
  + **sintaxe**: #pragma omp parallel for schedule(guided,chunk-size)
  + Chunk size is dynamic while using guided method, the size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads, and the size will be decreased to chunk-size (but the last chunk could be smaller than chunk-size).

**Exercício 1:**

1. A ordem de execução não é sempre a mesma, pois depende do escalonamento feito para cada uma das threads e como é claro, varia de execução para execução.
2. Não percebo muito bem o que se pretende com esta questão. O output dentro de uma thread é sempre sequencial, embota nem sempre a quantidade de outputs seja sempre o mesmo. (ver melhor)
3. Cada thread executa uma 100 vezes o ciclo.

**Exercício 2:**

1. Em, 2.1, cada thread processa metade dos ciclos T0 desde 0 .. 49 e T1 desde 50 .. 99.

Em 2.2, apenas a T0 executa o ciclo, conclusão a T0 é a master.

Em 2.3, A primeira thread disponível executa todas as iterações do ciclo;

1. Em, 2.1, sim, a divisão é sempre a mesma, cada thread processa metade dos ciclos T0 desde 0 .. 49 e T1 desde 50 .. 99.

Em 2.2, sim, apenas a T0 executa o ciclo, conclusão a T0 é a master.

Em 2.3, não, a primeira thread disponível executa todas as iterações do ciclo;

1. Não, porque em termos de concorrência, a thread que chega primeiro executa a iteração logo, nem sempre será a mesma thread a executar.

**Exercício 3:**

1. Sim, a execução é sempre a mesma. A sincronização é feita quando todas as threads atingem o ponto da barrier, isto é, todas as threads esperam pelas outras no ponto da diretiva barrier.
2. Sim, primeiro executa a T0 toda e só depois a T1 também toda.

...

#pragma omp parallel num\_threads(2)

#pragma omp for ordered

for(int i=0; i<100;i++) {

int id = omp\_get\_thread\_num();

#pragma omp ordered

printf("T%d:i%d ", id, i);

**Exercício 4:**

1. **4.1** 🡪 Como as diretivas *#pragma omp parallel for*, *schedule(static)* e *schedule(static, 1)* são perfeitamente iguais não se esperam alterações relativamente ao exercício 2.1. já quando se usa a diretiva *schedule(static, 10)* estamos a criar chunks com tamanho 100/10. Como temos duas threads a executar cada uma vai ficar com 10 chunks que serão atribuídos **por ordem** e então temos a T0 a executar as iterações 0 .. 9 e a T1 10 .. 19 ...  
   **4.2** 🡪 não, nem sempre são as mesmas threads a executar os mesmos chunks, pois no método dinâmico os chunks são atribuídos sem qualquer ordem. Com a diretiva ***schedule(dynamic,10)*** como seria de esperar se anteriormente a ordem não é a mesma agora também não poderia ser.  
   **4.3** 🡪 A resposta é a mesma que no ponto 4.2, pois no caso *guided* a única coisa que se altera é o tamanho do chunk que aqui é dinâmico.
2. Respondido na alínea a).  
   **4.1** 🡪 Sim  
   **4.2** 🡪 Não.  
   **4.3** 🡪 Não