PRÁTICAS | PROGRAMAÇÃO PARALELA | PL 4

→ vamos usar para correr o programa em paralelo

OpenMP → significa "open multiprocessing". É uma API ⊛ e é um padrão (permite) para a programação paralela de "shared Memory".

→ através da existência de threads

⊛ API → "application programming interface"

↓

é um conjunto de normas que possibilita a comunicação entre plataformas através de uma série de padrões e protocolos.

↓

é um fio de execução e basicamente é a entidade que vai executar algum código em paralelo.

Se tivermos 2 fios de execução, significa que existe uma parte da aplicação que é executada em paralelo com a outra parte da aplicação

Os fios só vão uma abstração ao software para indicar as tarefas que podemos executar em paralelo aos cores físicos ??

??

fazem parte do software (mas não vão uma coisa física) e cada thread vai indicar um caminho que o código pode seguir, ou seja, cada thread corresponde a um core.

FLAG - fopenmp → permite que possamos executar o código em paralelo (o código, assim, vai ser otimizado com paralelismo) ou compilado

PL4 → Exercício 1

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

int main() {
    printf("master thread \n");
# pragma omp parallel num_threads(2)
    for (int i=0; i<100; i++) {
```

isto vai ser aplicar exatamente ao código que está imediatamente a seguir

vai imprimir isto antes do ciclo for

isto pragma indica o nº de threads a usar utilizadas

⊛2 → como só incrementa no final do ciclo, o i vai até ao 100!!

o i vai ser de 0 até 99

→ cada vez que o ciclo acaba, vai-se incrementar

chamada da função, ou seja, o id é o nº da thread e vai ser 0 ou 1 (porque só temos 2 threads)

```
int id = omp_get_thread_num();
printf("T%d: i%d", id, i);
}
printf("master thread\n");
}
```

mas isto não faz sentido porque estamos a utilizar temos que utilizar 1 ou não mesmo?

Q2 → Se em vez de colocarmos 2, colocarmos 0 ou 1, o programa só vai executar uma thread, que é a thread 0, ou seja, não vamos executar um paralelo (a thread 0 vai obter os 100 resultados).

→ iriamos usar apenas a master thread !!

→ Como quando executamos um paralelo, cada fio vai executar 50 e por isso o tempo de execução deveria variar para metade

↓

mas isso não acontece !!

A ordem de "output" não é sempre a mesma em várias execução pois como estão 2 fios em execução não conseguimos garantir que os fios estão a executar ao mesmo instante. Daí podemos dizer que temos que fazer algum controlo para garantir que vão ao mesmo tempo.

A ordem de "output" de cada thread é sempre a mesma, isto é, dentro de cada thread a ordem das instruções mantém-se =, fazendo com que a ordem de "output" seja sempre a mesma.

→ A execução do loop não é distribuída (isto é, "scheduled") entre as threads porque estamos a fazer com que cada fio faça 100, e não é o que nós pretendemos (ou seja, não está haver distribuição nenhuma carga, que era o que nós queríamos).

PL4 → Exercício 2

*pragma omp → é a funcionalidade básica de comunicar informação ao compilador de C/C++ de modo a este gerar código otimizado para o ambiente de execução do OpenMP.
compilado

⚠ As diretivas seguintes devem ser introduzidas entre o *pragma omp parallel e o ciclo "for"!!
temos que inserir no código e tem que ser antes do ciclo "for"

*pragma omp for → atribuição de iterações de loop a threads. Basicamente, aqui já vamos ter distribuição de carga (neste exemplo):
- a thread 0 vai fazer de 0 a 49
- a thread 1 vai fazer de 50 a 99

} a divisão do loop é sempre a mesma, apesar da execução!!

Porém, continuamos com um problema de anteriormente: o "output" depende da thread que chegar 1º à região paralela
↓
Não esquecer que dentro de cada thread a ordem das instruções é a mesma.

*pragma omp master → restringe um bloco de código a ser executado apenas na "master thread".
ou mestre
Neste exemplo, aqui, como a thread 0 é a "master thread", faz tudo sozinha até ao fim (aqui não há distribuição de carga porque basicamente estamos a mandar só utilizar a "master thread").
→ só vamos ter uma thread a ser executada, a "master thread".
↓
como só temos uma thread, então, basicamente, o "output" vai ser sempre o mesmo apesar do nº de execuções

* pragma omp single → restringe um código de bloco a ser executado por uma única thread.

Neste exemplo, aqui, como anteriormente, não vamos ter distribuição de carga porque também só estamos a mandar executar uma única thread.

Aqui, ao contrário da diretiva anterior na qual só mandava executar a "master thread", aqui no single, ele manda executar a 1ª thread que chegar à região variável (neste caso, é a thread 0, mas noutros casos pode não ser).

* pragma omp critical → restringe a execução de um bloco de código a uma única thread em um tempo.

Neste exemplo, as 2 threads fazem 100 iterações cada uma, isto é, aqui também não há distribuição de carga!

aqui a thread 0 faz tudo até ao fim (até às 100 iterações) e só depois a thread 1 faz tudo até ao fim também (até às 100 iterações).

Basicamente, enquanto uma thread estiver dentro do "critical", as outras vão ser colocadas em espera, ou seja, aqui vai existir um controlo de fluxo. Nesmo mandando executar várias vezes, o "output" vai ser sempre o mesmo!! igual

PL 4 → Exercício 3 @3 → as threads 0 e 1 vão executar alternadamente as mesmas iterações.

* pragma omp barrier → faz com que todas as threads numa equipa esperem pelas restantes.

Neste caso, executa-se o ciclo "for" uma vez para cada thread, isto é, a thread 0 faz uma vez o ciclo e depois antes de fazer outro, a thread 1 realiza o ciclo, e assim sucessivamente!

@3

→ neste pragma, a ordem do "output" não é sempre a mesma pode variar (aqui pode ser a thread 0 a começar ou a thread 1)

NOTA: este pragma tem que ser incluído no loop, depois do printf.

tipo uma coisa de estafetas

Basicamente, temos 2 fios de execução e uma barreira: o que está acontecer é que os fios de execução estão a executar o loop e não ficar à espera dos fios seguintes (neste caso, fio) para poder continuar a realizar

#pragma omp ordered → especifica um bloco de código numa região do loop que vai ser executado na ordem das iterações do loop.

→ quem faz isto é o #pragma omp for ordered

Neste caso, à partida (ao início do ciclo), vai ser feita a distribuição 50/50 para cada thread; as iterações e/ou o ciclo não vão ser executados por ordem

↳ quem faz isto é o #pragma omp ordered

↓

Aqui fazemos a distribuição de carga por omissão.

NOTA: temos que colocar #pragma omp ordered dentro do loop e antes do printf; temos que adicionar a palavra "ordered" à diretiva #pragma omp for.

PL4 → Exercício 4

NOTA: aqui apenas vamos adicionar à diretiva #pragma omp for as seguintes palavras:

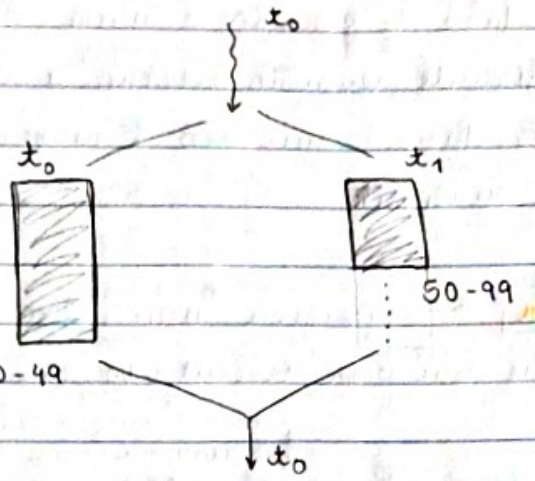
- schedule (static) → a distribuição do loop é sempre a mesma → a carga computacional (que é aquilo que temos de executar: neste exemplo, é o ciclo "for") está a ser dividida (50 para a thread 0 e 50 para a thread 1), ou seja, é feita uma divisão entre as threads das iterações / cargas (50/50), porém com intervalos irregulares (4) (o que é feito by default / por omissão), os quais vão igualmente apresentados em "output".

↳ os "outputs" variam porque depende de que thread vai primeiro!!

⚠ Antes de executar o ciclo, faz-se logo a divisão da carga (antes de executar o ciclo, os fios já sabem o que vão executar)

= schedule (static)

A divisão estática vai deixar de ser útil quando existem iterações + demoradas do que outras. (Nesse caso, uma das threads iria suportar + carga do que a outra



#4 → para melhor isto, podemos usar o schedule (static, 10), que neste exemplo, vai dividir a carga em blocos de 10, isto é, avança 10 até executar 100: a thread 0 fica com cargas 0 a 9, 20 a 29, e assim sucessivamente; a thread 1 fica com os intervalos no meio.

Em ambos os casos, vamos ter 50 iterações para cada thread e a ordem do print / output varia.

→ a distribuição do loop é sempre a mesma

- schedule (dynamic) → é feita uma divisão de cargas entre as threads mas de forma desconhecida e irregular, sempre que uma thread estiver disponível, vai pedir ao escalonador + cargas

↓

responsável pelo momento em que cada thread começa e/ou acabava!!

#5 → para é ser irregular, podemos usar o schedule (dynamic, 10), que, neste exemplo, existe uma divisão inicial, em 10 iterações para a thread 0 e outras 10 iterações para a thread 1 e quando uma delas acaba primeiro, vai lhe ser atribuída novamente + 10 iterações. Como o escalonador é indica os intervalos de carga, logo, entre as threads, pode é ter exatamente 10 cargas.

⚠ Quando executamos várias vezes, verificamos que os blocos de iterações não são sempre atribuídos às mesmas threads.

- schedule (guided) → semelhante à anterior mas o tamanho da divisão em blocos diminui durante a execução.