Insper

SuperComputação Igor Montagner, Luciano Soares 2020/1

09 - Tarefas

O uso de tarefas (tasks) para paralelizar códigos apresenta bons resultados em diversas aplicações que não se encaixam exatamente no modelo *fork-join*. Nessa aula exploraremos o uso de tarefas para paralelizar a execução de funções recursivas.

Parte 0 - comandos task e master

Revisão

- #pragma omp task cria uma tarefa, que será executada por alguma das threads disponíveis.
- #pragma omp master executa o bloco abaixo da diretiva na thread principal (id == 0)

Tarefa 1

Escreva um programa usando tarefas (tasks) que irá aleatoriamente gerar uma das duas cadeias de caracteres:

```
I think race cars are fun
I think car races are fun
```

Dica: use tarefas para imprimir a parte indeterminada da saída (ou seja, as palavras "race" ou "cars").

Pergunta 1

O programa retorna sempre a mesma resposta?

Essa situação é chamada de "Condição da corrida".



Uma **condição de corrida** ocorre quando o resultado de um programa depende de como o sistema operacional escalona as threads. Ou seja, seu resultado não é determinístico.

Parte 1 - tarefas e funções recursivas

O código abaixo calcula a sequência de Fibonacci

[https://pt.wikipedia.org/wiki/Sequ%C3%AAncia_de_Fibonacci] usando um algoritmo recursivo.

```
#include <iostream>

int fib(int n) {
    int x,y;
    if(n<2) return n;
    x=fib(n-1);
    y=fib(n-2);
    return(x+y);
}

int main() {
    int NW=45;
    int f=fib(NW);
    std::cout << f << std::endl;
}</pre>
```

Tarefa 2

Crie uma nova função int fib_par1(int n); e paraleliza as chamadas recursivas usando task s. Faça com que cada chamada recursiva seja executada como uma tarefa. **Dica**: é preciso esperar as tarefas acabarem antes do return?

Tarefa 3

Compare com o código original. Houve melhora? Por que?

O exercício acima exemplifica o custo de criar e escalonar tarefas. Vamos melhorar esta paralelização agora limitando o número de tarefas criadas.

Tarefa 4

Crie uma nova função int fib_par2(int n) e paraleliza as chamadas recursivas. Faça com que sejam criadas no máximo max_threads tarefas. **Dica**: passar como argumento o nível da recursão pode ajudar.

A etapa final de nossa aula trabalhará o algoritmo de cálculo do *pi* usando integração numerica novamente, mas agora escrito de maneira recursiva.



Pergunta 2

Abra o arquivo *pi_recursivo.cpp* e examine seu conteúdo. Quantos níveis de recursão são feitos? Em outras palavras, quantas chamadas são necessárias até que o for seja executado sequencialmente? **Dica**: veja a relação entre MIN_BLK e num_steps.

Tarefa 5

A paralelização de código é sempre muito mais fácil quando eliminamos todos os efeitos colaterais. Elimine todas as variáveis globais do código. Crie também uma função double pi_par_tasks(long num_steps) que chama a função double pi_r com os valores iniciais corretos.

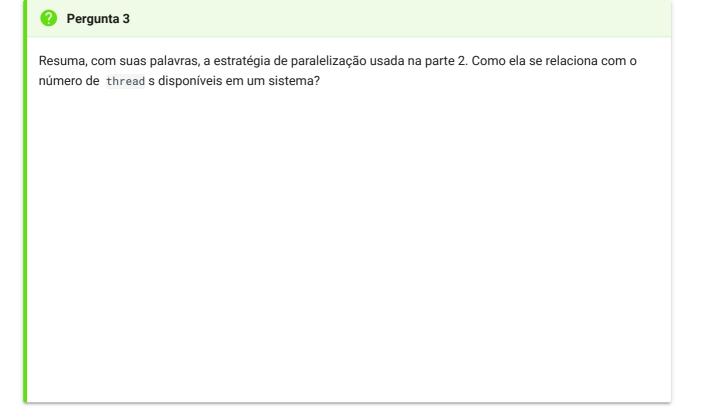
Tarefa 6

Agora use omp task para paralelizar as chamadas recursivas.

Tarefa 7

Varie o valor de MIN_BLK e meça o desempenho do programa. Escreva, junto desse valor, quantas tarefas foram criadas e quantos processadores estiveram ativos durante a execução do programa.

Parte 3 - estratégias de paralelização



Pergunta 4

Resuma, com suas palavras, a estratégia de paralelização usada na parte 3. Qual a relação entre MIN_BLK e o número de thread s disponíveis em um sistema? Devemos modificar MIN_BLK baseado na entrada?