*PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS*

*Instituto de Ciências Exatas e Informática*

*Departamento de Ciência da Computação - Curso de Ciência da Computação*

*Disciplina: Seminários II Segundo semestre de 2019*

*Professor: Saulo Augusto de Paula Pinto —* [***saulo@pucminas.br***](mailto:saulo@pucminas.br)

***Terceiro Roteiro***

***Seções críticas e balanceamento de carga***

No roteiro anterior, vimos um exemplo de uso de uma seção crítica no qual o tempo de execução da versão paralela do algoritmo ficou muito pior que o da versão sequencial. A pergunta que surge, então, é: será que seções críticas podem ser usadas de modo eficiente?

Veremos um exemplo cuja resposta é positiva. Vamos usar mais um exemplo (modificado) disponibilizado pelo Professor Luís Fabrício W. Góes. Seja o código sequencial que conta o número de (números) primos menores que um valor (dois milhões, no caso):

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

int ehPrimo(long num) {

long divisor;

if( num <= 1 )

return 0;

else

if(num > 3){

if(num % 2 == 0)

return 0;

long max\_divisor = sqrt(num);

for(divisor = 3; divisor <= max\_divisor; divisor += 2)

if (num % divisor == 0)

return 0;

}

return 1;

}

int main() {

long LIMITE\_MAX = 2000000;

long contaPrimo;

long soma;

int n;

if (LIMITE\_MAX <= 1)

soma = 0;

else {

soma = 1;

for (n = 3; n < LIMITE\_MAX; n += 2){

contaPrimo = ehPrimo(n);

soma = soma + contaPrimo;

}

}

printf("Número total de primos: %ld\n", soma);

return 0;

}

Você pode precisar *linkar* a biblioteca math.h no gcc. Compile assim:

gcc primos.c –o primos –lm

Rode o código.

Vamos paralelizá-lo. Ele tem dois laços (*loops*). *Normalmente, tentamos paralelizar o laço mais externo, pois a quantidade de trabalho nele é maior.* Insira a linha

#pragma omp parallel for private (contaPrimo) reduction(+:soma)

Antes da linha

for (n = 3; n < LIMITE\_MAX; n += 2){

A paralelização é quase igual à do programa de cálculo do Pi, que fizemos no roteiro anterior

Rode o programa e compare o tempo de execução.

Vamos usar uma seção crítica em vez da redução. Altere o código assim:

#pragma omp parallel for private (contaPrimo)

for (n = 3; n < LIMITE\_MAX; n += 2){

contaPrimo = ehPrimo(n);

#pragma omp critical

soma = soma + contaPrimo;

}

Rode o programa. Como ficou o tempo em relação à versão anterior? Para responder, pense no trabalho de cada *thread* na paralelização do laço “do n” em relação ao trabalho de contagem dos primos...

Agora que você já respondeu à pergunta, vamos tratar do balanceamento de carga. O OpenMP provê algumas opções (‘políticas”) para este balanceamento entre as *threads*, por meio da cláusula *schedule(política, chunkSize)*. “política” pode ser *static*, *dynamic*, *guided* ou *auto*. Além disso, é possível especificar o número de iterações para cada *thread* por meio do parâmetro *chunckSize*. O escalonamento *default* é o estático. Assim, teste os escalonamentos dinâmico e guiado com um *chunk* de 100, por exemplo. A diferença básica entre os escalonamento dinâmico e guiado é que este último varia o *chunkSize* até chegar no valor do parâmetro, mas o primeiro usa *chunks* do tamanho do parâmetro. Ambos vão distribuindo os blocos de iterações à medida que as *threads* terminam o *chunk* anterior.

#pragma omp parallel for private (cont\_primo) reduction(+:soma) schedule (dynamic,100)

for (n = 3; n < LIMITE\_MAX; n += 2){

contaPrimo = ehPrimo(n);

soma = soma + contaPrimo;

}

Tente variar as políticas e o tamanho do *chunk* para conseguir um *speedup* melhor. Nas próximas tarefas vamos tentar otimizar o balanceamento.