Instituto Tecnológico de Aeronáutica

CE-265: Processamento Paralelo

Exercício 03: Paralelizar Crivo de Eratostenes em OpenMP

Bruno Benjamim Bertucci - Turma 23.2

1 Introdução

O denominado Crivo de Eratostenes é um algoritmo para determinar a quantidade de números primos menores que um determinado limite N. Consiste em, inicialmente, marcar todos os números como se fossem primos, e, a partir do número 2, eliminar todos os seus múltiplos, e em seguida avançar na sequencia de números até que seja encontrado um número que ainda não foi eliminado, denominado base, sendo nessa primeira iteração o número 3. Esse procedimento de eliminação de múltiplos de um número primo, e avanço na sequência até o próximo número que não foi eliminado, que é um número primo, é repetido até que o limite N seja atingido.

Na implementação computacional do Crivo, algumas otimizações foram aplicadas. Por exemplo, só os números ímpares foram considerados, pois sabe-se de antemão que nenhum dos números pares maiores que dois são primos. Quando efetua-se a eliminação de múltiplos a partir de uma base qualquer, inicia-se pelo quadrado dessa base, pois todos os múltiplos anteriores a esse valor já foram eliminados por uma base anterior. Finalmente, A busca por novos números primos se encerra quando a raíz de N é atingida, pois os quadrados dos próximos números excedem N e, portanto, não podem ser considerados como bases.

2 Implementação da paralelização OpenMP

O programa que executa o Crivo de Eratostenes foi implementado na linguagem C. O componente principal desse programa, que efetua de fato a contagem dos números primos, foi separado em uma função denominada "Crivo". Dentro dessa função, o *OpenMP* foi utilizado para paralelizar os trechos onde tal otimização é permitida. O código final dessa função está no Anexo A.

A estrutura geral dessa função é constituída por um laço *do...while* para repetir o avanço da base, dentro do qual existem dois laços *for*, um que realiza a eliminação dos múltiplos de uma base a partir do quadrado dela, e outro que determina a próxima base, e interrompe o laço assim que a encontra.

O primeiro laço *for* pode ser paralelizado, pois não há ocorrência de condição de corrida dado que cada iteração depende somente do seu índice. Já o segundo laço *for* não pode ser paralelizado, pois o comando *break* interrompe a execução do laço. Em geral, portanto, somente a região do primeiro *for* pode ser paralelizada, o que foi feito usando a diretiva do *OpenMP*:

```
#pragma omp parallel for
```

3 Teste de tempos de execução do programa

A partir da implementação descrita, dois testes foram realizados. Em ambos, o programa foi executado para $N = 10^9$ repetidas vezes, usando 1, 2, 4, 8, 12, 16, 20, e 24 *threads*. No primeiro teste, a compilação foi feita sem nenhuma *flag* de otimização, enquanto no segundo teste, a *flag* de otimização "O3"foi introduzida para o compilador.

Os tempos de execução da função Crivo medidos para cada um dos testes, bem como os valores de *Speed-up* correspondentes (Equção 1, onde p é o número de processadores usado e $T(n)$ é o tempo de execução para uma quantidade n qualquer de processadores), estão na Tabela

1. Os gráficos contidos na Figura 1 ilustram a variação do tempo de execução com a variação do número de *threads*.

$$S(p) = \frac{T(1)}{T(p)} \tag{1}$$

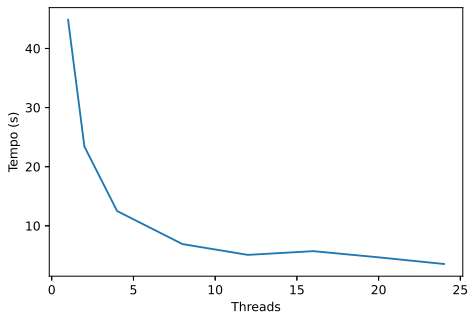
Threads	Sem -O3	Com -O3
1	44.842078	7.583083
2	23.408858	4.109065
4	12.498593	2.821831
8	6.910685	2.645273
12	5.082420	2.580490
16	5.714802	2.879838
20	4.668305	2.147177
24	3.544141	1.994288

(a) Tempos de execução.

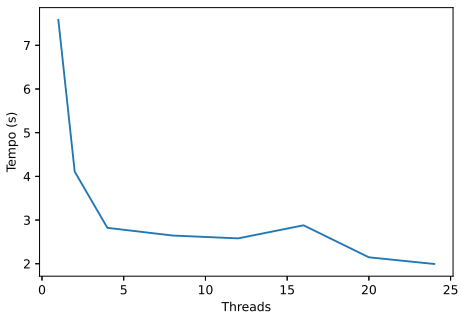
Threads	Sem -O3	Com -O3
2	1.915603	1.845452
4	3.587770	2.687292
8	6.488804	2.866654
12	8.822978	2.938621
16	7.846655	2.633163
20	9.605644	3.531652
24	12.652453	3.802401

(b) Valores de Speed-up.

Tabela 1: Resultados dos testes do programa.



(a) Tempos de execução sem a *flag* -O3



(b) Tempos de execução com a *flag* -O3

Figura 1: Gráficos dos tempos de execução.

Observando as tabelas e os gráficos, pode-se observar que essa implementação não escala seu desempenho bem com o número de *threads*. Usando 8 *threads* ou mais, o *speed-up* passa a se distanciar significativamente dos valores ideais, que seriam iguais ao número de *threads* usado. Isso é corroborado pelo texto do programa, que mostra que uma parte significativa do código não é paralelizado. Assim, conforme o número de *threads* aumenta, e a porção paralelizada passa a demandar menos tempo, a porção sequencial passa a ser responsável por uma parcela cada vez maior do tempo de execução total, configurando, assim, um limite para os ganhos da paralelização feita.

Um fato interessante a ser notado é que, em ambos os experimentos, o tempo de execução usando 16 *threads* é maior do que com 12 *threads*. Finalmente, observa-se que a *flag* de otimização "O3" usada no segundo experimento reduziu significativamente o tempo de execução do programa, porém, conforme o número de *threads* aumentou, a diferença de tempo com relação ao mesmo teste feito com o mesmo número de *threads*, mas sem a *flag*, foi reduzindo consistentemente. Em outras palavras, o distanciamento entre o *speed-up* medido e o seu valor ideal cresceu mais rapidamente com o aumento do número de *threads* quando a *flag* de otimização foi usada.

Anexo A – Código da função Crivo com paralelização *OpenMP*

```
// Crivo: encontra os primos ate ind2num(indMax)


void Crivo(int *primos, long indMax) {
  long sqrtMax;
  long indBase;
  long base;
  long i;

  // raiz quadrada do ultimo inteiro eh o maior fator primo

  sqrtMax = (long) sqrt((long) ind2num(indMax));

  // primeira base

  indBase=0;
  base=3;

  // para todas as bases

  do {

    // marca como nao primo os multiplos da base a partir do seu quadrado
    #pragma omp parallel for
    for (i=num2ind(base*base); i<=indMax; i+=base)
      primos[i]=0;

    // avanca a base para o proximo primo

    for (indBase=indBase+1; indBase<=indMax; indBase++)
      if (primos[indBase]) {
        base = ind2num(indBase);
        break;
      }

    // enquanto base nao superar o final de primos e
    //               nao superar a raiz quadrada do ultimo inteiro

  }
  while (indBase <= indMax && base <= sqrtMax);
}
```