UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
SISTEMAS DE INFORMAÇÃO

INE5645 - Programação Paralela e Distribuída
Profº Dr. Odorico Machado Mendizabal

Alan Vinicius Cezar Ensina (16206354)

A.3 Avaliação de desempenho

A análise a seguir mostra uma comparação de desempenho de um código sequencial e outro paralelo. Ambos os códigos são responsáveis em identificar todos os números primos dentro de um determinado conjunto de números.

Para mais informações sobre o código acesse o repósitório no github: https://github.com/alanensina/INE5645-ParallelAndDistributedProgramming/tree/master/Tarefa2

TEMPO DE PROCESSAMENTO

Para calcularmos o tempo de processamento ambos os códigos foram instrumentados com a biblioteca <*sys/time.h*> para fazer o cálculo de processamento.

No código sequencial, a contagem do tempo de processamento começa a contar antes do início do laço de validação dos números (linha 42) e finaliza a contagem após o laço finalizar (linha 46):

```
36  void verificaPrimosSequencial(int faixa) {
37
38    printf("------------------------------------------------------------\n");
39    printf("Números primos no intervalo de 1 a %d:\n", tam);
40    printf("------------------------------------------------------------\n");
41
42    gettimeofday(&t1, NULL);
43    for (int i = 1; i <= faixa; i++) {
44      validarNumero(i);
45    }
46    gettimeofday(&t2, NULL);
47
48    printf("------------------------------------------------------------\n");
49    printarTempoDeExecucao();
50    printf("------------------------------------------------------------\n");
51  }
```

Já no código paralelo a contagem de tempo é feita onde é definida a granularidade do código, ou seja, no laço onde são criadas as *threads* (linha 81 e 102):

```
81  gettimeofday(&t1, NULL);
82  for (int i = 0; i < cores; i++) {
83
84    Data * dados = (Data * ) malloc(sizeof(Data));
85
86    if (i == 0) {
87      dados -> min = array[intervalo * i];
88      dados -> max = array[intervalo * (i + 1)];
89    }
90
91    if (i != 0 && i < cores && i != cores - 1) {
92      dados -> min = array[(i * intervalo) + 1];
93      dados -> max = array[((i * intervalo) + 1) + intervalo - 1];
94    }
95
96    if (i == cores - 1) {
97      dados -> min = array[(i * intervalo) + 1];
98      dados -> max = array[tam - 1] + 1;
99    }
100   pthread_create( & threads[i], NULL, processar, (void * ) dados);
101 }
102 gettimeofday(&t2, NULL);
```

Após feito essa instrumentação no código, ambos os códigos foram executados por três vezes e feito uma média no tempo de processamento de cada um deles.

Como parâmetro, foi definido analisar os números primos de de 1 a 10.000.

Segue abaixo o tempo de processamento:

Sequencial: 0,007690 s
Paralelo (2 threads): 0,000160 s
Paralelo (4 threads): 0,000295 s
Paralelo (8 threads): 0,000640 s
Paralelo (16 threads): 0,001164 s
Paralelo (32 threads): 0,002152 s

Para verificarmos outro ponto de vista, foi definido analisar os números primos de de 1 a 5.000, ou seja, metade do problema anterior.

Segue abaixo o tempo de processamento:

Sequencial: 0,004948 s
Paralelo (2 threads): 0,000139 s
Paralelo (4 threads): 0,000229 s
Paralelo (8 threads): 0,000520 s
Paralelo (16 threads): 0,000980 s
Paralelo (32 threads): 0,001914 s

SPEEDUP E EFICIÊNCIA

O speedup é definido com a relação entre o makespan do programa sequencial com o makespan do programa paralelo com n núcleos de processadores:

$$Speedup(n) = \frac{tempo\_sequencial}{tempo\_paralelo(n)}$$

Já a eficiência é definida com o aproveitamento do paralelismo da plataforma:

$$Eficiencia(n) = \frac{Speedup(n)}{n}$$

Sendo assim, podemos definir o speedup e a eficiência do código testado da seguinte maneira:

Para uma amostra de 1 a 10.000:

n	**2**	**4**	**8**	**16**	**32**
S	48,06	2,61	12,01	6,60	3,57
E = S/n	24,03	0,65	1,50	0,41	0,11

Para uma amostra de 1 a 5.000:

n	**2**	**4**	**8**	**16**	**32**
S	35,60	21,61	9,51	5,05	2,58
E = S/n	17,8	5,40	1,19	0,31	0,08

Como podemos observar, conforme aumentamos o número de threads a eficiência foi caindo em ambos os casos, porém, curiosamente com 8 threads numa amostra de 10.000 números, obteve-se uma eficiência de 1,5, sendo assim a segunda maior da análise dessa amostra.

Já numa amostra de 5.000 números a eficiência teve uma linha mais consistente, onde conforme o número de threads aumenta, a eficiência diminui.

Sendo assim, podemos concluir que no problema em questão, é melhor optarmos em executar o programa conforme a eficiência seja acima de 1, pois assim teremos um tempo de processamento menor que o do programa sequencial.