UNIVERSIDADE FEDERAL DE SÃO CARLOS

CAMPUS SOROCABA

COMPUTAÇÃO PARALELA

PROJETO 1 - PTHREADS E NÚMEROS PRIMOS

Érico Alexandre Nielsen Matthiesen | 400556

Gabriel Stankevix Soares | 511340

Maurício Spinardi | 408174

Setembro de 2015

SUMÁRIO

1. O EXPERIMENTO

Uma linha ou encadeamento de execução (thread) é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrencialmente. Uma thread permite, por exemplo, que o usuário de um programa utilize uma funcionalidade do ambiente enquanto outras linhas de execução realizam outros cálculos e operações. Muito útil para evitar comportamentos bloqueantes e incrementar o desempenho de aplicações com as características adequadas. Em hardwares equipados com uma única CPU, cada thread é processada de forma aparentemente simultânea, pois a transição entre threads ocorre de forma transparente e tão rápida que, para o utilizador, isso ocorre paralelamente. Em hardwares com múltiplos CPUs ou multinúcleos, no entanto, threads são realizadas, de fato, de forma simultânea.

Nesse experimento, discutiremos as possibilidades de se utilizar threads para incrementar o desempenho de uma aplicação serial que calcula um determinado número de primos. Serão exibidas as transformações efetuadas no código serial para que se comporte como um programa paralelo, bem como os cálculos de *speedup* e considerações sobre *tradeoff* com diferentes núcleos e regiões de corrida concorrente.

1.1. CÓDIGO SERIAL

Como base do experimento de paralelização, assumimos uma única solução serial para o problema de encontrar n números entre os primeiros primos. Considere o arquivo de extensão .C disponível e o trecho abaixo:

```
for ( count = 2 ; count <= n ;  ) {
      for ( c = 2 ; c <= i - 1 ; c++ ) {
         if ( i%c == 0 ) break;
      }
      if ( c == i ) {
         printf("%d\n",i);
         count++;
      }
      i++;
   }
```

Podemos notar no fragmento acima o trecho de maior custo computacional desse código serial. É nele onde os n primeiros números primos são, de fato, calculados. Esse é, portanto, o trecho alvo de paralelização nesse código serial.

1.2. PROPOSTA DE SOLUÇÃO E CÓDIGO PARALELO

Considere a possibilidade de identificar se um número é primo ou não. Agora, verifique se dois números diferentes são primos, da mesma forma e ao mesmo tempo. Essa é a base da solução paralela proposta nesta seção:

```
	void* is_prime() {
	while (global_iterator < primes) {

		pthread_mutex_lock(&lock);
			number += 1;
			int my_number = number;
		pthread_mutex_unlock(&lock);

		int c;
		for ( c = 2 ; c < my_number; c++ ) {
			if ( my_number%c == 0 )		break;
		}
		if ( c == my_number ) {
			pthread_mutex_lock(&lock);
				int i = global_iterator;
				global_iterator++;
			pthread_mutex_unlock(&lock);

			primes_found[i] = my_number;
		}
		my_number++;
	}
	return (void**)0;
}
```

Note, no fragmento acima, que existem duas regiões de acesso concorrente. Em ambas ocorre o mesmo: um número é consumido pela thread e o seguinte é gerado, para que a próxima thread que esteja disponível possa consumi-lo. Uma vez consumido um número, a solução é exatamente a mesma proposta pelo código serial, ou seja, uma sequência de divisões por força bruta. Se um número é identificado como primo, então ele é salvo em um vetor de resposta, em uma posição específica que somente a thread em questão tem acesso. Essas regiões de acesso/escrita concorrente requerem uma análise de *tradeoff* cuidadosa, uma vez que quão maior for o número de threads, maior deverá ser o empate durante a execução.

Por fim, vale notar que, como números diferentes são testados em paralelo, não é possível determinar a ordem em que serão identificados como primos ou descartados. Essa é a situação que exige o uso do vetor resposta (que aqui chamamos de *primes_found*). Para exibir o resultado em ordem crescente, uma forma de contornar o problema decorrente da paralelização se dá utilizando o vetor de resultados, que pode ser ordenado através de um algoritmo conhecido de ordenação, como o quicksort padrão da linguagem utilizada (C).

1.3. SPEEDUPS E CONSIDERAÇÕES

Para verificar a diferença de tempo do algoritmo com paralelização e o serial, realizamos 3 testes para eles encontrarem 50.000, 100.000 e 150.000 números primos com 2,4 e 8 threads no algoritmo paralelo.

Todos os teste foram realizados em um computador com memoria de 4.0 GiB e um processador Intel® Core™2 Duo CPU E7500 @ 2.93GHz × 2. Todos os tempos nos testes estão em segundos.

50.000 primos	**Teste 1**	**Teste 2**	**Teste 3**	**Media**
Serial	75	75	76	75.333
2 Threads	25	26	25	25.333
4 Threads	25	25	26	25.333
8 Threads	25	25	26	25.333

100.000 primos	Teste 1	Teste 2	Teste 3	Media
Serial	325	325	326	325.333
2 Threads	106	108	107	107
4 Threads	107	107	107	107
8 Threads	107	107	108	107.333

150.000 primos	Teste 1	Teste 2	Teste 3	Media
Serial	755	756	755	755.333
2 Threads	251	251	251	251
4 Threads	250	251	251	250.666
8 Threads	249	250	250	249.666

Ao observar os resultados dos testes e comparar os tempos, percebemos que o numero de threads teve pouca influencia no tempo final do algoritmo, pois o ganho de tempo com o numero maior de threads é perdido no tempo de processo da região critica.

Então, para a realização dos cálculos de *speedup* de um programa paralelo, utilizaremos a seguinte formula:

S = T serial / T paralelo

Onde T serial é o tempo de execução serial e T paralelo é o tempo de execução paralelo.

Teste 50.000 primos onde a média do tempo serial é igual 75.333

p	2	4	8
S	2.97371018	2.97371018	2.97371018

Teste 100.000 primos onde a média do tempo serial é igual 325.333

p	2	4	8
S	3.040495327	3.040495327	3.031062208

Teste 150.000 primos onde a média do tempo serial é igual 755.333

p	2	4	8

S	3.009294821	3.013304557	3.0253739

Agora calcularemos a eficiência do programa paralelo utilizando a seguinte formula:

*E = S / p = (T seria / T paralelo) / p = T serial / p*T paralelo*

Onde p é o numero de core, T serial é o tempo de execução serial e T paralelo é o tempo de execução paralelo.

Teste 50.000 primos

p	**2**	**4**	**8**
E = S / p	1.48685509	0.743427545	0.371713773

Teste 100.000 primos

p	**2**	**4**	**8**
E = S / p	1.520247664	0.760123832	0.378882776

Teste 150.000 primos

p	**2**	**4**	**8**
E = S / p	1.504647411	0.753326139	0.378171738

2. CONCLUSÃO E DIFICULDADES

A realização deste projeto proporcionou a chance do primeiro contato prático com códigos de execução paralela. O decorrer da experiência permitiu que identificássemos com relativa facilidade o trecho em que threads poderiam ser implementadas a fim de otimizar a execução do algoritmo, no entanto, permitiu também que enfrentássemos as primeiras dificuldades em como transpor uma solução serial para modalidade paralela. Ordenar o resultado obtido pela execução do código paralelo foi um desafio, não sendo possível transpor a solução serial de forma natural: na versão paralela, os resultados são obtidos desordenadamente e são reorganizados por completo, no final. Por fim, um último desapontamento foi identificado: os bloqueantes de acesso concorrente se provaram indispensáveis para a solução proposta, minando o desempenho que seria ganho com o aumento no número de cálculos ocorrendo em paralelo.