UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Thierry Pierre Dutoit

Trabalho 1 de Computação Concorrente:

Crivo de Eratóstenes e Computação Concorrente

Professora: Silvana Rossetto
Data: 04/2021
Local: Rio de Janeiro

Crivo de Eratóstenes

Eratóstenes, matemático grego nascido pouco antes de Cristo, desenvolveu um método para encontrar ou contar primos que, posteriormente, obteve como nome Crivo de Eratóstenes. A ideia central do método é separar os números compostos dos números primos, guardando os primos e descartando os compostos.

Para aplicar o método precisamos definir um inteiro N que será nosso valor limite de contagem. Em seguida, define-se uma lista inteiros de tamanho (N-1)/2 com valor igual a 1 para todos os campos. Com a lista inicializada, pegamos o primeiro valor ímpar >= 3 não nulo, digamos f (possível primo), e zeramos os elementos de f em f. Repetimos esse processo até encontrar um valor de f no qual $f^2 > n$ e paramos. Como resultado, teremos uma lista de primos, como mostra o exemplo a seguir:

Supondo N = 40:

3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39

Selecionamos f = 3 e zeramos de f em f:

3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39

3 5 7 **0** 11 13 **0** 17 19 **0** 23 25 **0** 29 31 **0** 35 37 **0**

Selecionamos o próximo f não nulo, logo f = 5 e repetimos:

3 **5** 7 0 11 13 0 17 19 0 23 25 0 29 31 0 35 37 0

3 **5** 7 0 11 13 **0** 17 19 0 23 **0** 0 29 31 0 **0** 37 0

Ao selecionar o próximo f, vemos que $f^2 = 49 > 40$ e paramos. Ficamos então com a lista de número primos de 2 até n:

2 3 5 7 11 13 17 19 23 29 31 37

Configuração da Máquina Usada nos Testes

Hardware

Arquitetura:	x86_64
CPU op-mode(s):	32-bit, 64-bit
CPU(s):	4
Thread(s) per core:	1
Cores(s) per socket:	4
Socket(s):	1
Processor:	Intel(R) Core(™) i5-3330 CPU @ 3.00GHZ
L1d cache:	128 KiB
L1i cache:	128 KiB
L2 cache:	1 MiB
L3 cache:	6 MiB

Software:

Linux Version: 5.8.0-50-generic(buildd@lgw01-amd64-030) (gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #56~20.04.1-Ubuntu SMP Mon Apr 12 21:46:35 UTC 2021

GCC Version: gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0

Pseudocódigo do Crivo de Eratóstenes

Em [1] (pág. 64 e 65) temos o seguinte pseudocódigo descrito:

Entrada: inteiro positivo ímpar n.
Saída: lista de primos ímpares <= n.

Etapa 1: Comece criando um vetor v de (n-1)/2 posições, cada uma das quais deve estar preenchida com o valor 1; e fazendo P = 3.
Etapa 2: Se P^2 > n escreva os números 2j + 1 para os quais a j-ésima entrada do vetor v é 1 e pare; senão vá para a Etapa 3.
Etapa 3: Se a posição (P - 1)/2 do vetor v está preenchida com 0 incremente P de 2 e volte à Etapa 2; senão vá para a Etapa 4.
Etapa 4: Atribua o valor P^2 a uma nova variável T: substitua por zero o valor da posição (T-1)/2 do vetor v e incremente T de 2P; repita estas duas instruções até que T > n; quando isto acontecer incremente P de 2 e volte à Etapa 2.

Crivo de Eratóstenes Sequencial

A implementação do pseudocódigo para um programa em c está em crivosequencial.c, e podemos testar duas coisas: desempenho e corretude. Para desempenho, aumentaremos a entrada de N e, para a corretude, podemos contar o número de primos, em vez de listá-los, e verificar com outras fontes se o programa está correto (pois não temos ainda outro meio de comparação). Script usado: myscript.

Sequencial	Tempo de Execução	Menor Tempo	Número de Primos
10^5 Elementos	0,000419	0,000276	9592
	0,000319		
	0,000357		
	0,000276		
	0,000403		
10^7 Elementos	0,081292	0,079573	664579
	0,084143		
	0,079573		
	0,081664		
	0,084066		
10^9 Elementos	11,028694	11,028694	50847534
	11,620046		
	12,197006		
	11,883446		
	12,084156		

O objeto de análise por enquanto está na corretude do nosso programa sequencial, portanto, analisaremos apenas a quantidade de primos encontrados. Como comparação, usaremos [2], e vemos que:

100,000	9,592
1,000,000	78,498
10,000,000	664,579
100,000,000	5,761,455
1,000,000,000	50,847,534

Como os valores estão corretos podemos prosseguir para a implementação concorrente do Crivo de Eratóstenes.

Crivo de Eratóstenes Concorrente

Para implementar o paralelismo de dados no nosso problema é necessário definir como dividir o trabalho para as threads. Esta parte é provavelmente a mais difícil em questão, visto que existem muitas maneiras. Inicialmente, adotei como estratégia intercalar os fatores entre as threads. Ou seja, se uma thread começa com fator 3, a próxima começará com o ímpar não nulo seguinte, sendo 5 nesse caso, e assim adiante. Essa estratégia foi escolhida pela limitação de ferramentas que podem ser adotadas neste trabalho, visto que outras estratégias envolvem implementações mais complexas e robustas.

Então, a forma de implementação da nossa estratégia consiste em passar como argumento para as threads os fatores iniciais. De acordo com o cálculo (2 * nthreads) elas saberão o próximo fator a ser pego, e repetirão o processo até encontrar um fator maior que raiz(n).

Para a implementação teremos como estruturas de dados compartilhadas a quantidade de elementos, o número de processos e o vetor que armazena os números primos encontrados. Cada thread recebe como argumento o fator inicial que será usado para cortar seus múltiplos, e elas não retornam valor pois só estão encarregadas de eliminar os valores compostos do vetor. O nome do programa que implementa essa estratégia é crivoconcorrente.c.

Casos de Teste

O teste da corretude do programa pode ser feito assim: para números de elementos diferentes comparamos a quantidade de primos encontrados pelo programa concorrente e pelo programa sequencial. Podemos fazer isto pois já testamos a corretude do programa sequencial e já fizemos os testes dele.

Testando o programa concorrente para N = 10^5, 10^7 e 10^9, Número de Threads = 1, 2, 3 e 4 temos os resultados (myscript2):

Conc. 1 Thread	Número de Primos	Conc. 2 Threads	Número de Primos
10^5 Elementos	9592	**10^5 Elementos**	9592
10^7 Elementos	664579	**10^7 Elementos**	664579
10^9 Elementos	50847534	**10^9 Elementos**	50847534
Conc. 3 Threads	**Número de Primos**	**Conc. 4 Threads**	**Número de Primos**
10^5 Elementos	9592	**10^5 Elementos**	9592
10^7 Elementos	664579	**10^7 Elementos**	664579
10^9 Elementos	50847534	**10^9 Elementos**	50847534

Em comparação com o programa sequencial vemos que os resultados estão certos:

Sequencial	Número de Primos
10^5 Elementos	9592
10^7 Elementos	664579
10^9 Elementos	50847534

Uma suposição válida seria que o programa só funciona para entradas múltiplas de 10. Por mais que não seja verdade, não é relevante também, pois todos os testes serão feitos em torno destes valores. Isto se dá pelo custo de tempo de verificação de outros valores, já que precisamos de referências comparativas e não existem muitas disponíveis (não que eu tenha encontrado ao menos).

Avaliação de Desempenho

O conjunto de casos testes para avaliação do desempenho será:

N = 10^5, 10^7 e 10^9
Thread(s) = 1, 2, 3 e 4.

Com estes valores teremos uma boa comparação de desempenho entre o programa concorrente e o programa sequencial. As configurações da máquina já foram mencionadas, porém de forma breve, temos 4 núcleos (por isso escolhemos até 4 threads, mais que isso perderíamos desempenho pelo overhead de criação de threads), um processador Intel i5 terceira geração e S.O Linux.

Para cada combinação de N e Thread(s) fizemos 5 testes, e observamos o menor valor gerado. No entanto, vale a pena observar o tempo médio também para ter uma visão geral do desempenho obtido:

Conc. 1 Thread	Tempo de Execução	Menor Tempo	Tempo Médio
10^5 Elementos	0,00069	0,000591	0,000657
	0,000613		
	0,000657		
	0,000591		
	0,00068		
10^7 Elementos	0,086512	0,083186	0,086512
	0,092493		
	0,083186		
	0,095112		
	0,085842		
10^9 Elementos	10,78239	10,78239	10,850966
	10,80538		
	11,863499		
	10,850966		
	10,913856		

Conc. 2 Threads	Tempo de Execução	Menor Tempo	Tempo Médio
10^5 Elementos	0,000782	0,000501	0,000681
	0,00066		
	0,000681		
	0,000501		
	0,000719		
10^7 Elementos	0,062142	0,057201	0,062142
	0,06219		
	0,057201		
	0,066015		
	0,060089		
10^9 Elementos	7,717488	7,717488	7,837551
	7,777593		
	8,417795		
	7,837551		
	8,200271		

Conc. 3 Threads	Tempo de Execução	Menor Tempo	Tempo Médio
10^5 Elementos	0,000489	0,000432	0,000451
	0,000447		
	0,000451		
	0,000432		
	0,000477		
10^7 Elementos	0,055231	0,051195	0,055231
	0,055422		
	0,051195		
	0,055865		
	0,05411		
10^9 Elementos	6,870788	6,870788	7,472558
	7,213914		
	7,847974		
	7,472558		
	7,748414		

Conc. 4 Threads	Tempo de Execução	Menor Tempo	Tempo Médio
10^5 Elementos	0,000554	0,000453	0,000497
	0,000477		
	0,000497		
	0,000453		
	0,000548		
10^7 Elementos	0,051727	0,048611	0,051727
	0,051782		
	0,048611		
	0,051812		
	0,04971		
10^9 Elementos	6,931589	6,931589	7,001642
	6,961484		
	7,671834		
	7,001642		
	7,103686		

Comparando com o programa sequencial, que já foi mostrado anteriormente:

Sequencial	Tempo de Execução	Menor Tempo	Tempo Médio
10^5 Elementos	0,000419	0,000276	0,000357
	0,000319		
	0,000357		
	0,000276		
	0,000403		
10^7 Elementos	0,081292	0,079573	0,081664
	0,084143		
	0,079573		
	0,081664		
	0,084066		
10^9 Elementos	11,028694	11,028694	11,883446
	11,620046		
	12,197006		
	11,883446		
	12,084156		

Podemos calcular o ganho (aceleração) de cada caso teste, fazendo o cálculo Tsequencial/Tconcorrente. Não consideramos aqui tempo de inicialização e finalização pois não fazem parte da seção crítica de execução do programa. No entanto, normalmente o programa concorrente toma mais tempo nestas etapas por ter mais estruturas. Então, temos a seguinte tabela que mostra a aceleração do programa concorrente em relação ao programa sequencial:

10^5 Elem.	Tempo Médio	Ganho
Sequencial	0,000357	1
1 thread	0,000657	0,543378995
2 threads	0,000681	0,524229074
3 threads	0,000451	0,791574279
4 threads	0,000497	0,718309859

10^5 Elem.	Menor Tempo	Ganho
Sequencial	0,000276	1
1 thread	0,000591	0,467005076
2 threads	0,000501	0,550898203
3 threads	0,000432	0,638888888
4 threads	0,000453	0,609271523

10^7 Elem.	Tempo Médio	Ganho
Sequencial	0,081664	1
1 thread	0,086512	0,943961531
2 threads	0,062142	1,31415146
3 threads	0,055231	1,478589922
4 threads	0,051727	1,578749976
10^9 Elem.	Tempo Médio	Ganho
Sequencial	11,883446	1
1 thread	10,850966	1,095150975
2 threads	7,837551	1,516219288
3 threads	7,472558	1,590278189
4 threads	7,001642	1,69723702

10^7 Elem.	Menor Tempo	Ganho
Sequencial	0,079573	1
1 thread	0,083186	0,956567210
2 threads	0,057201	1,391112043
3 threads	0,051195	1,554311945
4 threads	0,048611	1,636934027
10^9 Elem.	Menor Tempo	Ganho
Sequencial	11,028694	1
1 thread	10,78239	1,022843173
2 threads	7,717488	1,429052303
3 threads	6,870788	1,60515708
4 threads	6,931589	1,591077313

Discussão Dos Testes

O ganho de desempenho alcançado não foi o esperado por dois motivos: há condição de corrida no nosso programa e não conseguimos lidar com a parte mais crítica de processamento do problema.

1. **Condição de Corrida:** Adotamos a estratégia de alternância entre fatores das threads, e isso pode ser problemático. Um exemplo, se uma thread recebe como fator 3 e outra thread recebe como fator 9, a ordem de execução delas é muito importante. Se a primeira executar e eliminar o 9 da lista de primos, a segunda thread irá pular para o próximo fator. Caso a segunda thread execute primeiro, ambas as threads eliminarão múltiplos de 3, desperdiçando tempo. Isto é um mal necessário, pois não é o intuito deste trabalho usar mecanismos de sincronização para ordenação de execução das threads.

2. **Parte Crítica de Processamento:** Não pegamos a seção mais crítica do programa pela falta de instrumental disponível. É claro que, mesmo com a condição de corrida, temos ganho de desempenho com alternância de fatores entre as threads. Porém, o real gargalo do problema é na eliminação dos múltiplos destes fatores. Para implementar essa estratégia precisamos adotar outra estratégia, que divide o trabalho de eliminação dos valores compostos, e que requer um ferramental maior.

Além dos pontos mencionados acima, há o problema de acesso à memória. Este sendo um dos piores fatores e totalmente limitado às especificações de cada máquina (se o programa for bem implementado).

Melhorias a serem feitas

Podemos mudar a estratégia adotada para a divisão da tarefa entre as threads. Em vez de alterarmos os fatores entre as threads, podemos ter um processo raiz que define o próximo primo a ser escolhido e elimina os múltiplos da sua parte do vetor. As outras threads ficam encarregadas apenas de eliminar os múltiplos, e o vetor fica dividido em partes iguais a n/p (p = processo) para um bom balanceamento de carga.

O programa crivoconcorrente3.c foi implementado com a estratégia mencionada acima, porém não mostrou o ganho de desempenho esperado devido a alguma falha de implementação (uma questão de memória e/ou de demora com a ordenação das threads). Neste programa eu usei exclusão mútua, fugindo do intuito deste trabalho e sendo resultado de minha curiosidade. Uma implementação dessa estratégia pode ser encontrada em [3], com uma melhor explicação, mas faz uso de funções e constantes que desconheço resididas em uma biblioteca implícita chamada “mpi.h”.

Refinando o programa crivoconcorrente3.c notei um ponto muito importante. O tempo de desempenho do meu programa estava sendo afetado pelo acesso à memória. Apenas duas modificações já mostraram resultados diferentes:

typedef long long int ll -> typedef unsigned int
ll *v -> char *v

Outras mudanças foram necessárias para suportarem as duas linhas modificadas acima, porém ao usar menos memória o programa(crivo_otimizado.c) obteve um bom ganho de desempenho, como os testes abaixo mostram em myscript4 (N = 10^9 e Thread(s) = 4):

Não Otimizado	Otimizado
9,522855	5,274594
9,523187	5,206539
9,461414	5,308484
9,34941	5,313278
9,230189	5,330689

Mesmo que haja ganho de desempenho, o programa ainda está sendo limitado pela memória pois para um número de threads menor (em específico 3 na minha máquina) o desempenho do programa otimizado consegue ser maior. Isso se dá pois as memórias caches são razoavelmente pequenas na minha máquina e o programa depende bastante de cache hit para um bom desempenho.

Como todos os testes feitos na seção AVALIAÇÃO DE DESEMPENHO não buscavam otimização de memória, é interessante refazer os testes brevemente para observar um ganho de tempo de execução maior:

O programa crivoconcorrente.c para N = 10^9 e 4 Threads leva em média, agora otimizado, 4.8 segundos para terminar. Em relação ao nosso programa sequencial feito inicialmente no trabalho temos um ganho de aproximadamente 2,5. Visto todas as limitações de memória aqui constatadas, para um maior ganho de desempenho seria necessário pensar em outra implementação que use menos memória, porém minha máquina já parece estar bem limitada para maiores ganhos de desempenho. Além da questão de acesso à memória, temos os pontos mencionados na seção de DISCUSSÃO.

Então, fico feliz com o desempenho que alcancei e foi muito interessante ver a relação de concorrência e memória presentes neste problema. Gostaria de ter realizado testes em uma máquina mais potente para analisar resultados mais sólidos, porém não tenho nenhuma máquina mais potente com o mesmo sistema operacional (Linux).

Referências:

[1] Coutinho, S.C. Números Inteiros e Criptografia RSA. Segunda Edição. Rio de Janeiro, IMPA, 2014.

[2] Caldwell, Chris K. How Many Primes are There? PrimePages, 2021. Disponível em: https://primes.utm.edu/howmany.html. Acesso em: 22 abril 2021.

[3] Botelho, Fabiano Cupertino. Projeto e Análise de Algoritmos, 2006. Disponível em: https://homepages.dcc.ufmg.br/~nivio/cursos/pa06/tp4/tp42/tp42.pdf. Acesso em 22 abril 2021.