Exercício de Programação 1 – Programação Paralela

Cálculo de Números Primos

Daniel Souza de Campos – 2018054664

UFMG – 2021/2

1. Introdução

Este trabalho tem como objetivo desenvolver dois algoritmos para encontrar a lista de primos até certo número máximo N. O primeiro algoritmo é sequencial, ou seja, é executado por apenas uma *Thread*/Processo e o segundo algoritmo deve ser paralelo, ou seja, deve fazer uso de 2 ou mais *Threads*/Processos.

O algoritmo base para encontrar a lista de primos até um número N é conhecido como O Crivo de Eratóstenes. Para o trabalho, esse algoritmo deve ser aprimorado e, depois, paralelizado.

2. Ferramentas para o trabalho e execução

O trabalho foi desenvolvido utilizando a linguagem C++ com o compilador GCC no editor de código *Visual Studio Code*. A máquina de desenvolvimento é um Dell 8GB RAM processador Intel core i5 7° geração com 4 processadores lógicos 2,7 GHz. O sistema operacional é Windows 10, mas possui o *Windows Subsystem for Linux* (WSL) instalado, o que permitiu o desenvolvimento utilizando OpenMP.

Como repositório e gerenciador de versões, foi utilizado o Github e o trabalho está disponível em https://github.com/Pendulun/ErastothenesPrimesParallel e ficará público após a data de entrega do trabalho dia 29/11/2021.

Após compilar via *make*, o programa *primes* gerado consegue alternar entre os algoritmos sequencial e paralelo de acordo com o número de Threads passado como argumento.

3. Algoritmo sequencial

O Crivo de Eratóstenes original é descrito pelos seguintes passos:

1. Crie uma lista de inteiros consecutivos de 2 até N
2. Seja $p = 2$, o primeiro primo
3. Enumere todos os múltiplos de p até N e marque-os como não primos na lista criada no primeiro passo. O número p deve continuar na lista pois ele é primo
4. Encontre o primeiro número na lista que seja maior do que p e que não esteja marcado como não primo. Caso não exista tal número, pare. Caso contrário, seja p igual a este número (que é o próximo primo), repita a partir do passo 3.
5. Quando o algoritmo terminar, os números não marcados presentes na lista serão todos os números primos entre 1 e n.

De imediato, percebe-se a necessidade de alocar N espaços de memória para realizar o algoritmo. Entretanto, de acordo com https://primes.utm.edu/howmany.html, é possível aproximar a quantidade de números primos até N pela seguinte equação:

$$M = \frac{N}{(\ln N) - 1}$$

Dessa forma, o algoritmo sequencial proposto não aloca N espaços de memória inicialmente, mas sim, o resultado de aplicar a equação acima sobre N em uma tentativa de ter uma abordagem mais *Memory Friendly*.

Segue o pseudocódigo para o algoritmo sequencial:

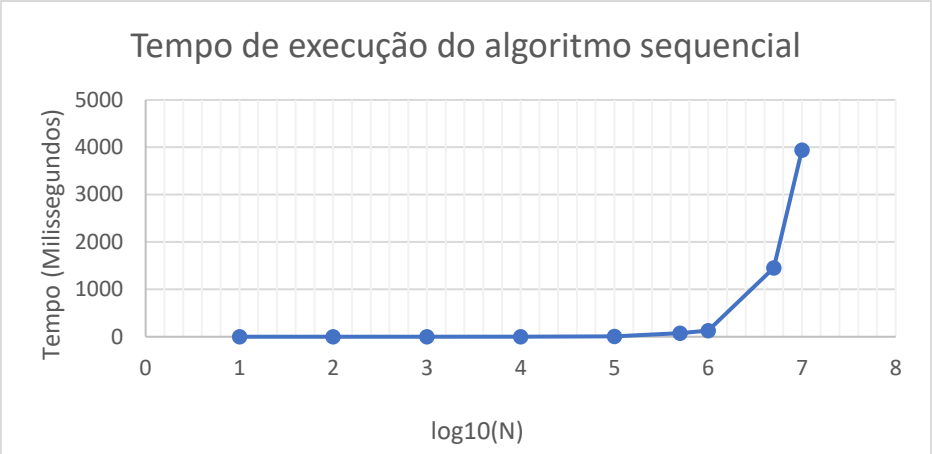
1. Aloque um vetor de tamanho M de acordo com a equação apresentada acima
2. Se N menor que 2, pare
3. Adicione ao vetor de primos de tamanho M o número 2
4. Se N igual a 2, pare
5. Para todo número ímpar x entre 2 e N faça:
    a. Para todo número primo y no vetor de primos menor ou igual a raiz quadrada de x, verifique se x é múltiplo de y.
    b. Se x não for múltiplo de nenhum dos y considerados, adicione x ao vetor de primos e continue para o próximo x.
    c. Se for múltiplo de algum y, continue para o próximo x.

Esse algoritmo possui três otimizações no total:

- Ele não aloca N espaços de memória, mas sim M
- Ele considera apenas números ímpares além do dois
- Como o vetor de números primos é naturalmente ordenado, consideramos apenas os z primeiros números que sejam menores ou iguais a raiz quadrada de um x.

3.1 Desempenho

Desempenho versão sequencial		
N	log10(N)	Tempo Médio 10 execuções (Milissegundos)
10	1	0
100	2	0
1000	3	0
10000	4	0.1
100000	5	7.7
500000	5.69897	76.1
1000000	6	126.8
5000000	6.69897	1454.8
10000000	7	3943.2



Apesar de bem rápido, pode-se perceber uma tendencia exponencial no tempo de execução proporcional à entrada N. O desempenho dele se deve ao conhecimento do problema, o que possibilitou a adicionar pontos de parada estratégicos na verificação de primalidade de um número como descrito na seção anterior.

4 Tentativa de Paralelização

Observando a implementação sequencial, pode-se inferir que existem dois pontos de possível paralelização: Os dois loops que compõem o algoritmo. O loop mais externo é extremamente dependente de dados, já que, para verificar se um dado número x é primo, devemos conhecer todos os primos que existem antes dele. O loop mais interno é perfeitamente paralelizável, já que uma iteração não depende da outra.

Seguindo essa ideia, foram desenvolvidas algumas variações de algoritmos paralelos utilizando o OpenMP na tentativa de melhorar o desempenho da verificação de primalidade de um número, mas apenas dois se mostraram promissores.

4.1 Primeiro algoritmo Paralelo

O primeiro algoritmo altera um pouco a verificação de primalidade de um número para o seguinte pseudocódigo:

1. Para todo número y entre 2 e a raiz de x:
    a. Verifique se x é múltiplo de y.
2. Se, em algum momento, encontramos um divisor de x, então ele não é primo. Se não encontramos, então x é primo.

Esse algoritmo, com certeza, é mais lento do que a verificação no sequencial pois ele considera todos os números entre 2 e a raiz de x e não somente os primos encontrados até x. Entretanto, esse processo pode ser facilmente paralelizado o que traria um ganho de desempenho no algoritmo.

Para paralelizar esse processo, foram usadas as diretivas *parallel* e *for* para dividir o trabalho da procura de um divisor de x no intervalo fixo de 2 até raiz de x. A ideia é que toda *Thread* salve se ela encontrou um divisor de x ou não. Se nenhuma Thread encontrou um divisor, então x é primo.

Segue os resultados da execução desse algoritmo:

Tempo (millisegundos) Médio de Execução 10 iterações Alg 1			
N	Num Threads		
	2	3	4

100	0	0	0
1000	3.3	1.1	1.5
10000	15.4	15.8	17
100000	181.4	175.1	190.1
1000000	2658.4	2830.6	2782
2000000	6845.1	7029.9	6564.2

Em comparação com o algoritmo sequencial, ele foi muito pior e também segue tendencia de tempo exponencial. A questão é que o *overhead* de criação e sincronização das Threads se tornou considerável pois fazemos a verificação para todo número ímpar entre 2 e N. Isso impactou diretamente na execução do algoritmo. Além disso, como não é possível "parar" as Threads se alguma delas encontrar um divisor antes de terminar todas as suas iterações, elas acabam testando divisores desnecessariamente.

4.2 Segundo algoritmo paralelo

Como dito anteriormente, a paralelização feita com o uso do OpenMP não permite que uma Thread saia de dentro de um loop antes de terminar o seu número fixo de iterações. Nesse ponto do trabalho, acreditava-se que essa restrição tenha contribuído para aumentar o tempo de execução do algoritmo paralelo 1. Dessa forma, uma outra implementação foi desenvolvida para tentar simular essa quebra de loop.

A segunda implementação segue a ideia de que, ao invés de procurar um divisor de x em todos os números entre 2 e a raiz de x, podemos realizar a procura ordenadamente em $h = teto\left(\ln\left(\sqrt{x}\right)\right)$ blocos que compõem esse espaço de verificação. O cálculo da quantidade de blocos envolve logaritmo para permitir que esse número cresça de acordo com o espaço de procura. Dessa forma, procuramos em cada um dos $M = \frac{\sqrt{x}}{h}$ blocos até que achemos um divisor ou até que os espaço de procura se esgote. Seguindo essa ideia, esperava-se que essa divisão em blocos atuasse como uma espécie de *break* na procura.

A procura no bloco em si continua sendo paralelizada pelas mesmas diretivas que no algoritmo 1. Seguem os resultados:

N	Tempo Médio Exec Alg 2 (millisegundos)		
	Num Threads		
	2	3	4
100	0	0.1	0
1000	1.5	2.2	2.1
10000	28.2	30.4	32.4
100000	300.1	316.3	337.8
1000000	3286.1	3285	3654.8
2000000	6762.7	6764.2	7820.9

Apesar de ter melhorado o tempo de execução em algumas ocasiões, não é possível afirmar que esse método ofereceu ganho de performance considerável em relação ao algoritmo 1, sendo pior, inclusive, em todos os tempos ao utilizar 4 Threads.

Pode ser que essa segunda implementação tenha diminuído o tempo de procura de divisores para vários números x, entretanto, para os números primos, deve-se verificar todo o espaço de procura, o que faz com que tenhamos ainda mais *overhead* de paralelização do que no algoritmo 1 para esses casos.

Dessa forma, apesar de ter desempenho inferior ao sequencial, o algoritmo 1 foi escolhido como o melhor entre os algoritmos paralelos propostos e o segundo algoritmo foi descartado, mas ainda pode ser visto nos commits como “algoritmo 4” no Github.

5.1 Alterando escalonamento do algoritmo paralelo

Seguindo a análise de performance para o algoritmo paralelo escolhido, foram testados vários cenários para identificar qual tipo de escalonamento seria o melhor. Seguindo a lógica do algoritmo paralelo 1, pode-se inferir que o custo de verificação de primalidade para um dado x não se altera a cada iteração, mas sim a cada x testado pois o seu espaço de procura é maior do que todos os outros x anteriores.

Tempo (millisegundos) Médio Execução 10 iterações (static, 4)			
N	Num Threads		
	2	3	4
100	0	0.1	0
1000	1	1.1	1.7
10000	14.9	15.8	16.9
100000	179.1	177.1	185.5
1000000	2571.1	2634.3	2576
2000000	6148.2	6307	5943.3

Tempo (millisegundos) Médio Execução 10 iterações (guided)			
N	Num Threads		
	2	3	4
100	0.1	0	0.3
1000	1	1.2	1.4
10000	14.3	16.4	17.2
100000	196	192.9	309.7
1000000	2586.2	2607.8	3186.7
2000000	5976.4	6040.2	7236.9

Tempo (millisegundos) Médio Execução 10 iterações (auto)			
N	Num Threads		
	2	3	4
100	0	0	0
1000	1.1	1.5	1.2
10000	15	15.8	16.9
100000	173.5	171.7	196.6
1000000	2473.6	2527.8	2530.2
2000000	6370	6031.2	5812.5

De acordo com os testes realizados, para o escalonamento *static*, ao variar o *chuncksize* entre 2 e 4, os tempos de execução mostraram uma tendencia de queda geral principalmente ao usar 4 Threads. No caso, estão sendo mostrados apenas os resultados para o *chuncksize* igual a 4.

Ao usar o tipo de escalonamento *guided*, ele se mostrou melhor em comparação com o *(static, 4)* de forma geral, sendo inferior apenas ao usar 4 Threads.

O tipo de escalonamento *auto* foi o que apresentou melhores resultados gerais a partir do uso de 3 Threads. Como esse tipo faz com que o próprio sistema escolha um tipo de escalonamento, provavelmente, ele usou o *static* com um *chuncksize* maior que 4 pois, como mencionado anteriormente, ao usar o tipo *static* e aumentando o tamanho do *chuncksize* a tendencia foi de queda no tempo de execução.

A performance para o escalonamento *dynamic* não foi mostrado pois ela se mostrou bem inferior a todos os outros.

6 Comparação entre os algoritmos sequencial e paralelo

Foram realizados testes para comparar o tempo de execução entre o algoritmo sequencial e o melhor dos algoritmos paralelos implementados com o escalonamento *auto*. Segue os resultados:

N	Tempo Médio 10 execuções (milissegundos)	
	Sequencial	Melhor paralelo
10	0	0
100	0	0
1000	0	1.2
10000	0.1	16.9
100000	7.7	196.6
500000	76.1	1493.1
1000000	126.8	2530.2
5000000	1454.8	18493.4
10000000	3943.2	45546.6

Como comentado ao longo do trabalho, a implementação do algoritmo paralelo não foi capaz de superar o desempenho do sequencial e, de fato, ficou muito inferior.

7 Conclusão

O objetivo do trabalho era implementar uma versão sequencial e outra paralela de um mesmo algoritmo para calcular os números primos entre 2 e um número de entrada N. Esse objetivo foi alcançado ainda que o tempo de execução do algoritmo paralelo tenha sido superior ao sequencial.

Infelizmente, a versão paralela se mostrou bem inferior em questão de performance se comparado ao sequencial. Isso se deve a abordagem realizada para construir o algoritmo sequencial com o objetivo de ser *Memory Friendly* e que utilizou do conhecimento do problema para interromper antecipadamente a verificação de primalidade de um número, o que aumentou a sua eficiência.

A forma na qual os algoritmos paralelizados foram desenvolvidos também tentaram levar em consideração esse conhecimento estratégico na procura, entretanto, o *overhead* do paralelismo, com certeza, foi o que mais influenciou no tempo de execução.

Apesar de tudo, foi possível por em prática os conhecimentos adquiridos durante as aulas sobre paralelismo e OpenMP além de mostrar que se faz necessário criatividade e prática para, realmente, paralelizar um processo com sucesso.