Crivo de Eratóstenes

Braian Melo e Leonardo Ribeiro

26 de outubro de 2025

1 Introdução

Este trabalho tem como objetivo implementar o algoritmo Crivo de Eratóstenes de duas maneiras: uma sequencial e outra paralela. Ao longo deste documento, iremos explicar como foi realizada a implementação de cada método, as decisões técnicas tomadas e os resultados obtidos na comparação entre os dois métodos.

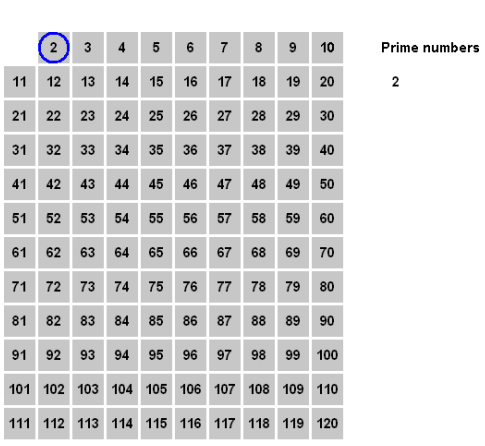
2 Crivo de Eratóstenes

Eratóstenes foi um filósofo pertencente ao mundo grego, bastante importante em sua época. Ele nasceu em Cirene e, ainda jovem, mudou-se para Atenas. Na fase adulta, Eratóstenes tornou-se diretor da Biblioteca de Alexandria, cunhou o termo "Geografia" e mediu a circunferência da Terra milênios antes de outros cientistas.

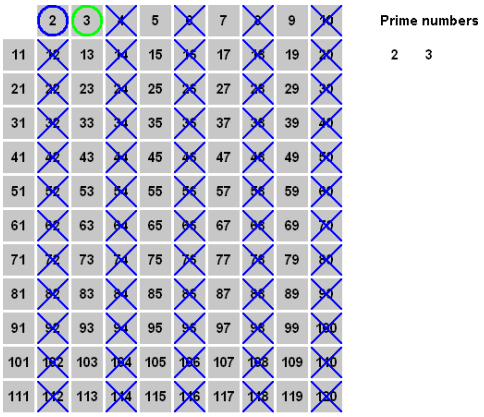
No entanto, seu trabalho mais citado no mundo da computação é o método que leva o seu nome: o crivo, ou prova, de Eratóstenes. Este método tem como objetivo encontrar todos os números primos de uma lista até um certo valor N. A prova é feita utilizando uma tabela com os números de 2 até N e funciona da seguinte forma:

1. Escolhe-se o primeiro número na tabela que não esteja marcado; no caso, o início sempre será o 2.
2. Marca-se todos os números que são múltiplos do número escolhido.
3. Repete-se o processo até a raiz quadrada de N.

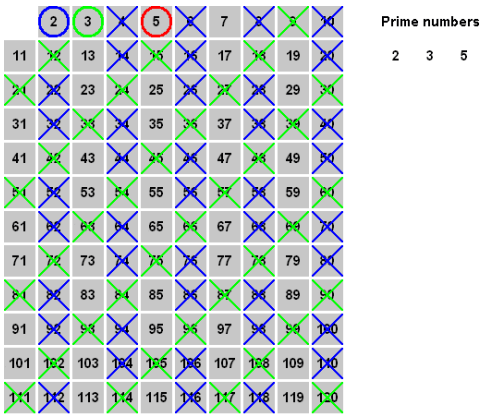
Abaixo está a representação visual do método:



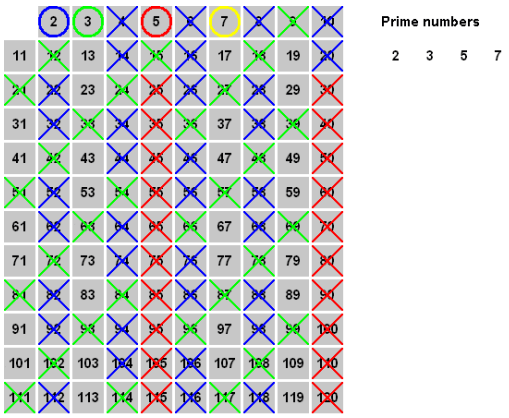
(a) Crivo de Eratóstenes: primeiro passo



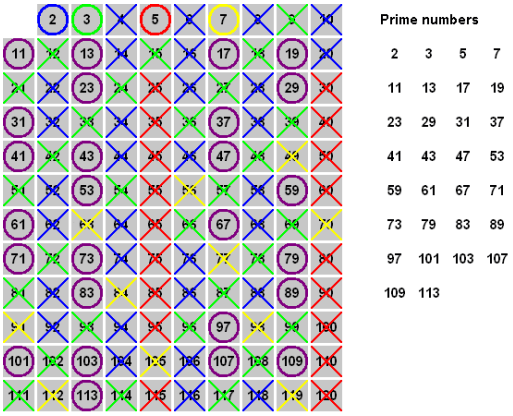
(b) Após marcar os múltiplos de 2, 3 é o próximo número livre.



(a) Agora os múltiplos de 5.



(b) Agora os múltiplos de 7.



(a) O resultado final com apenas os primos restantes.

3 Implementação

3.1 Requisitos e Entrada do Usuário

Para a execução do programa, são necessários alguns requisitos básicos. O sistema deve possuir o compilador `mpicc` instalado, bem como o ambiente de execução `MPI` configurado corretamente. Além disso, é recomendável que a máquina possua múltiplos núcleos de processamento para que o paralelismo seja efetivo.

O programa recebe como entrada o valor máximo N, que define o limite superior do intervalo de busca pelos números primos. Além disso, o usuário deve especificar o modo de execução: `s` para o modo sequencial e `p` para o modo paralelo.

Um exemplo de execução no terminal é apresentado a seguir:

```
$ mpirun -np 4 Programa.out 1000 p
```

Neste exemplo, o programa é executado com 4 processos (`-np 4`) para encontrar todos os números primos até 1000, utilizando o modo paralelo. Os resultados são gravados em um arquivo de saída chamado *primos.txt*.

3.2 Método sequencial

Na implementação do método sequencial, foi seguido o pseudocódigo encontrado em diversos materiais sobre o Crivo de Eratóstenes. O algoritmo pode ser descrito da seguinte forma:

```
1. Criar uma lista de números naturais não marcados de 2, 3, ..., n
2. k ← 2
3. Repetir:
   (a) Marcar todos os múltiplos de k entre k² e n
```

```
   (b) k ← menor número não marcado maior que k
   até que k² > n
4. Os números não marcados ao final são os números primos.
```

Para a implementação do algoritmo, foi necessário definir uma estrutura de dados para armazenar os números a serem analisados. Inicialmente, considerou-se o uso de uma lista encadeada, pois ela permitiria a remoção de elementos, os números não primos, de forma simples durante o processo. Entretanto, concluiu-se que essa abordagem seria ineficiente em termos de tempo, devido ao elevado custo de acesso a elementos não contíguos na memória.

Outra alternativa analisada foi a utilização de uma lista contígua com realocação, na qual os elementos seriam reordenados a cada eliminação. No entanto, essa estratégia também se mostrou custosa, pois a reorganização da lista após cada iteração impactaria significativamente o desempenho.

Diante dessas considerações, optou-se por uma lista contígua estática, em que cada posição armazena um número e um valor booleano indicando se ele foi marcado como não primo. Embora essa abordagem consuma mais memória, ela se mostrou mais eficiente em tempo de execução, visto que elimina a necessidade de remoções e realocações dinâmicas. Abaixo está o código referente a essa estrutura:

```
typedef struct bloco {
 int numero;
 bool marcado;
} Bloco;

typedef struct lista {
 int n;
 int qtd;
 int qtd_primos;
 Bloco *elementos;
} Lista;
```

3.3 Método paralelo

A partir da análise do método sequencial implementado, foram consideradas duas possíveis abordagens para a paralelização:

1. **Primeira abordagem:** paralelizar a marcação dos múltiplos de cada número não múltiplo de um primo descoberto. Por exemplo, na primeira iteração, o número 2 elimina todos os números pares. A partir disso, cada processo poderia ser responsável por verificar os números ímpares restantes. No entanto, essa abordagem se mostrou inviável, pois o algoritmo depende dos resultados das iterações anteriores. Assim, diferentes processos poderiam realizar operações sobre números que já foram invalidados em outras etapas.

2. **Segunda abordagem:** calcular todos os primos até $\sqrt{N}$, uma vez que para determinar se um número N é primo, basta verificar divisores até sua raiz quadrada. Após essa etapa, os processos paralelos são responsáveis por marcar os múltiplos desses primos dentro do intervalo $[\sqrt{N}, N]$, distribuindo a carga de trabalho entre os nós disponíveis.

Seguindo essa linha de raciocínio, optou-se pelo segundo método. Na implementação, apenas o processo mestre é responsável por calcular os números primos até $\sqrt{N}$. Em seguida, ele distribui entre os processos escravos uma lista contendo esses primos, dividida de forma igualitária. Cada processo escravo calcula os múltiplos apenas dos primos presentes em sua fatia do vetor. Ao final da execução, o processo mestre reúne os resultados parciais e armazena o conjunto completo de números primos em um arquivo.

3.4 Dificuldades

Durante a realização deste trabalho, algumas dificuldades foram enfrentadas. A mais comum, em programas paralelizados, foi a condição de corrida, quando alguns processos alcançavam certas

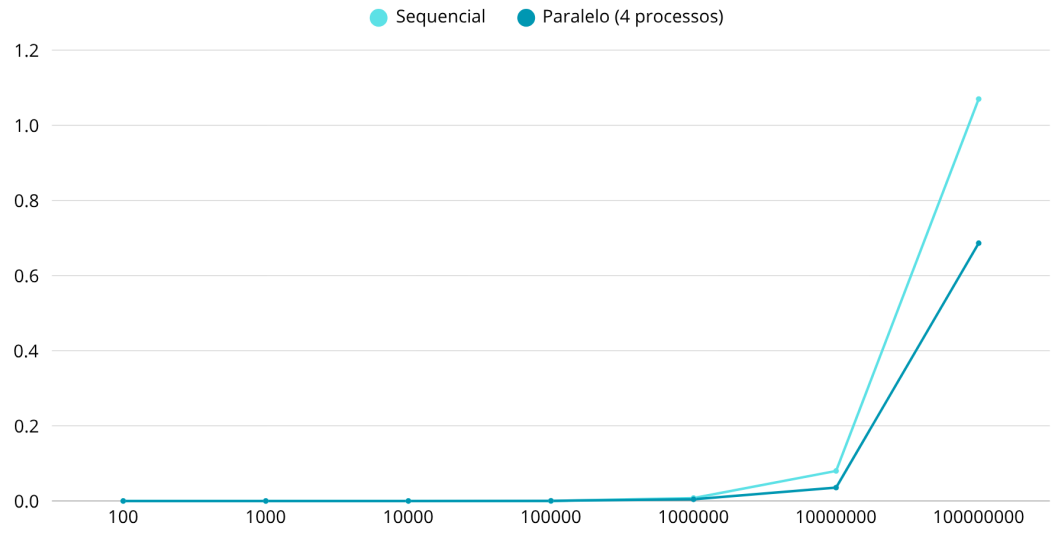
regiões de memória antes de receber os dados necessários. Para resolver esse problema, foram utilizadas barreiras, garantindo que todos os processos sincronizassem e permitindo que o código executasse corretamente.

Outra dificuldade que não chegou a ser totalmente resolvida foi a análise de vazamento de memória. O programa *Valgrind*, utilizado para essa análise, apontava vazamentos relacionados ao MPI, mas não ao código implementado, dificultando a interpretação dos resultados.

4 Resultados

A seguir, apresentam-se os gráficos de tempo obtidos. Para cada valor de **N**, foram realizadas cinco iterações, com o objetivo de medir o desempenho médio em cada caso.

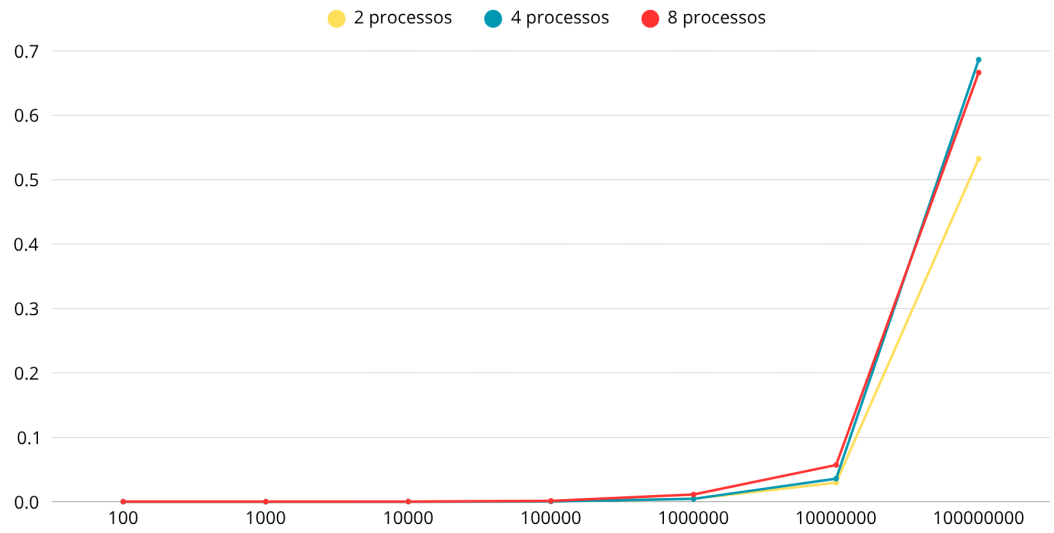
O gráfico a seguir compara a execução do programa em modo sequencial com a execução utilizando quatro processos. Devido à limitação do tipo de variável utilizado (*int*), não foi possível realizar testes com valores de **N** maiores. Entretanto, os resultados obtidos indicam que a paralelização, considerando seus custos, não apresenta vantagens significativas para valores pequenos de **N**. À medida que o tamanho da entrada aumenta, observa-se que a paralelização torna-se mais vantajosa.



(a) Comparação de tempo entre o método paralelo e o sequencial.

Outro ponto importante a ser destacado é que o programa utilizado para medir o tempo, o *gprof*, só registra tempos superiores a 0,01 s. Isso fez com que, para valores de **N** menores que 1.000.000, o tempo de execução no modo sequencial fosse arredondado para 0.

No próximo gráfico, apresentamos uma comparação entre diferentes quantidades de processos envolvidos. Observa-se que o tempo total de execução não diminui proporcionalmente ao aumento do número de processos. Devido à granularidade e ao custo de comunicação entre o processo mestre e os escravos, programas com mais processos podem apresentar maior lentidão. Contudo, também é possível inferir que, quanto maior o valor de **N**, mais vantajoso torna-se utilizar mais processos, tornando a execução mais rápida em comparação a valores menores de **N**.



(a) Comparação do tempo de execução com diferentes números de processos.

Como o gráfico, infelizmente, dificulta visualizar as diferenças entre os tempos nos casos de N pequeno, apresentamos abaixo uma tabela com todos os tempos medidos.

Tabela 1: Tempos de execução (em segundos) para diferentes números de processos.

N	Sequencial	2 processos	4 processos	8 processos
100	0,00000000	0,00003000	0,00000500	0,00009400
1.000	0,00000000	0,00004000	0,00001100	0,00011100
10.000	0,00000000	0,00010000	0,00004800	0,00028400
100.000	0,00000000	0,00060000	0,00044900	0,00128420
1.000.000	0,00783100	0,00459000	0,00445000	0,01118000
10.000.000	0,08000000	0,02953000	0,03572000	0,05699700
100.000.000	1,07000000	0,53252000	0,68628000	0,66627600

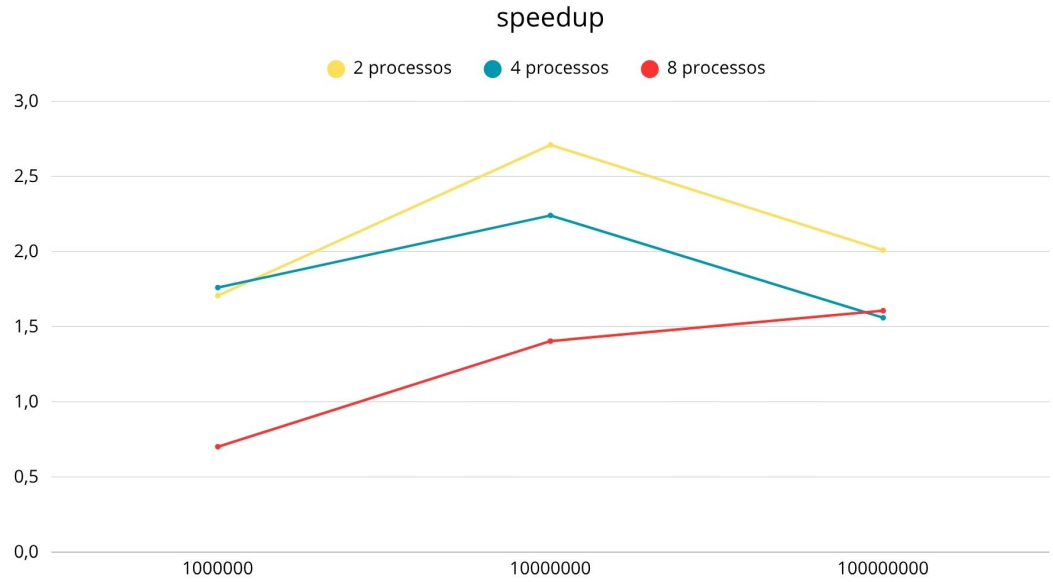
A partir desses resultados, é possível calcular o **speedup** (S) e a **eficiência** (E) da paralelização:

$$S = \frac{T_{seq}}{T_{par}} \tag{1}$$

$$E = \frac{S}{P} \tag{2}$$

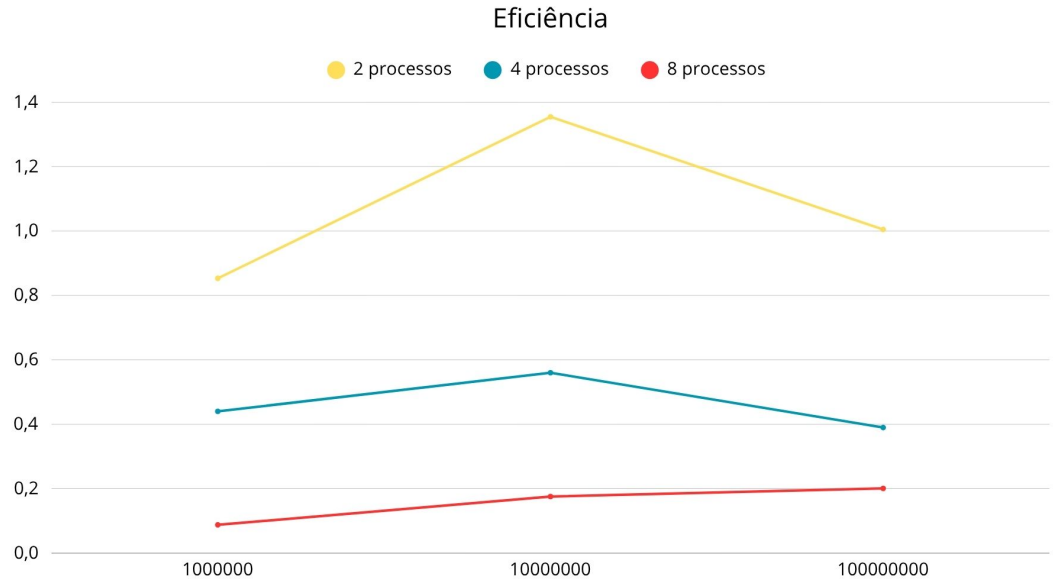
onde T_{seq} é o tempo de execução sequencial, T_{par} é o tempo de execução paralela com P processos, e P é o número de processos.

Abaixo, apresenta-se o gráfico de **speedup** dos testes. Observa-se que, ao rodar com 8 processos, o algoritmo inicialmente apresenta desempenho inferior ao sequencial, mas seu crescimento é quase linear. Isso sugere que, à medida que o tamanho do problema **N** aumenta, a configuração com 8 processos tende a se tornar a mais eficiente.



(a) Gráfico de speedup.

Em seguida, apresentamos o gráfico de **eficiência**. Nota-se que o maior aproveitamento ocorreu com apenas 2 processos paralelos. Entretanto, para $N = 100,000,000$, a eficiência dos casos com 2 e 4 processos diminui, enquanto a configuração com 8 processos continua a apresentar crescimento, indicando que, para problemas maiores, mais processos podem se tornar vantajosos.



(a) Gráfico de Eficiência.

Em resumo, a análise confirma que a paralelização traz benefícios significativos para problemas de maior escala, mas o número ideal de processos depende do tamanho do problema e do custo de comunicação entre eles.

5 Conclusão

Este trabalho implementou e comparou abordagens sequencial e paralela do Crivo de Eratóstenes utilizando MPI. Os resultados mostraram que, para valores pequenos de **N**, a paralelização não apresenta vantagens significativas devido aos custos de comunicação. À medida que **N** aumenta, a execução paralela se torna progressivamente mais eficiente, embora o aumento do número de

processos nem sempre reduza o tempo proporcionalmente. Assim, o estudo confirma que a paralelização traz ganhos de desempenho principalmente em cenários de maior escala, validando a importância da análise de custo-benefício em algoritmos paralelos.

Referências

SOUSA JUNIOR, J. G.; VIEIRA NETO, L. P. **Algoritmo paralelo para o Crivo de Eratóstenes**. 2009. Trabalho apresentado, Programação Concorrente e Paralela. Disponível em: ⟨https://www.inf.puc-rio.br/~noemi/pcp-13/primos.pdf⟩. Acesso em: 25 out. 2025.

CP-ALGORITHMS. **Sieve of Eratosthenes**. Disponível em: ⟨https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html⟩. Acesso em: 25 out. 2025.

BBC BRASIL. **O gênio africano que, há mais de 2 mil anos, com um graveto, provou que Terra é redonda**. 2023. Disponível em: ⟨https://www.bbc.com/portuguese/geral-64289025⟩. Acesso em: 25 out. 2025.