Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Programação Paralela — Exercício de Programação 2

Crivo de Erastótenes

Alexander Thomas Mol Holmquist

1 de agosto de 2021

1 Introdução

Neste trabalho exploramos o algoritmo "Crivo de Eratóstenes", para a geração de primos dentro do intervalo $[2, n]$, onde $n \leq 1e9$. Especificamente, buscamos uma implementação paralelizada através da interface Open MPI (Message Passing Interface) [1], distribuída gratuitamente para a linguagem C++. Esta interface provê comandos para facilmente criar e interagir com processos em um sistema de memória distribuída.

2 Desafios

2.1 Memória

O maior desafio para a implementação do Crivo de Eratóstenes, mesmo em sua versão sequencial, é o gasto de memória. Em sua implementação trivial, a complexidade de espaço é $O(n)$. Esse número é principalmente devido ao armazenamento de todos os números de 1 até n, nessa implementação. Rapidamente, tais limites assintóticos se mostram inviáveis.

Tendo isto em vista, optamos por manter somente uma "janela" dos números que estamos visitando. Essa janela possui tamanho **constante**, dependente dos sistemas em que o programa está rodando, e é representada na forma de um conjunto de bits (*bitset*), para maximizar a economia. O tamanho escolhido foi a metade do cache de nível 1. Desta forma, já desviamos de um obstáculo considerável. Utilizamos a implementação de *bitsets* provida pela biblioteca *boost* [2], pois permite alocação dinâmica do espaço ocupado pela estrutura, em contraste com a biblioteca padrão para C++, que só provê alocação estática para o *bitset*.

A próxima otimização para minimizar o uso de memória foi a utilização de vetores para armazenar os primos obtidos até o momento. Inicialmente, estávamos utilizando listas duplamente encadeadas, o que se provou extremamente ineficiente: constantes alocações na memória, e um gasto de 32 bytes (ao invés de 4) para armazenar cada elemento são os seus maiores defeitos. Na seção 3, mostramos um gráfico do desempenho do programa quando utilizávamos listas, com 2 cores de processamento (Figura 2).

2.2 Paralelização

A estratégia de paralelização do algoritmo é bem simples: cada processo é responsável por um segmento uniformemente distribuído da sequência de 1 a n. A primeira janela é calculada para todos os processos, para evitar uma perda de eficiência com sincronização logo no início da execução. Quando um processo Px precisa de novos primos que foram calculados por um processo Pk, fazemos uma sincronização entre Px e Pk, enviando os primos que Px precisa para continuar seus cálculos. Como a entrada deste processo específico está limitada a $1e9$, e a maioria das arquiteturas atuais contém caches de nível 1 de tamanho maior que 10KB ($(10000 * 4)^2 > 1e9$), só fazemos essa sincronização uma vez, no final. Não é difícil implementar o comportamento mais geral; esta restrição foi tomada para simplificar o projeto. A Figura 1 ilustra esta estratégia.

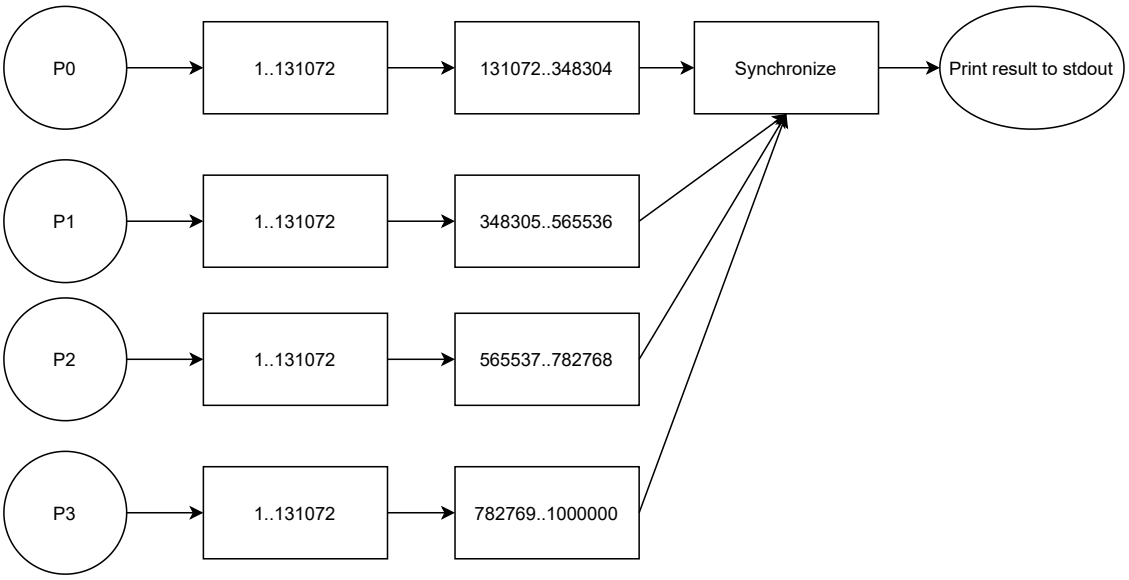

Figura 1: Estratégia de paralelização do algoritmo. A figura ilustra um exemplo com 4 unidades de processamento, e um problema de tamanho `1e6` (buscamos dar como saída todos os números primos entre `1` e `1e6`).

3 Avaliação

Mostramos tabelas com medidas de *speedup* e eficiência, para medir o ganho de poder de processamento ao pular de um para dois núcleos (Tabela 2). Além disso, mostramos uma comparação entre a implementação com lista, e utilizando vetor para armazenar os números primos (Tabela 1). A máquina utilizada possui somente dois núcleos de processamento[1]. Por isso, não foi possível estender o gráfico para verificar o comportamento posterior, ao aumentar ainda mais o número de núcleos.

Medimos o tempo de processamento para cada tamanho de entrada 10 vezes, e fazemos a média para calcular os tempos vistos nas tabelas. O script que gera os arquivos contendo os resultados é chamado de *testTimeBenchmarks.sh*. Um notebook *IPython* também foi criado, com o nome de *timePlots.ipynb*, para gerar os gráficos que podem ser vistos abaixo. Após as tabelas, apresentamos gráficos que ilustram o mesmo comportamento.

	Lista (n = 2)	Vetor (n = 1)	Vetor (n = 2)
Tempo de Exec. (s)	13.160096	9.526111	4.937262

Tabela 1: Resultados para tamanho `1e8`. Comparamos a implementação utilizando listas e vetores. Claramente, a implementação utilizando vetores é superior, em termos de tempo de execução. Além disso, note-se que o consumo de memória também foi diminuído em grande parte, simplesmente como consequência desta modificação. Veja as Figuras 1 a 3 para mais detalhes.

[1]Configurações da máquina: Intel Core i5-7200U CPU @ 2.50GHz. 1 processador físico; 2 núcleos; 4 threads.

N	Tempo (1p)	Tempo (2p)	Speedup	Eficiência
1e1	0.000115	0.000096	1.205021	0.602510
1e2	0.000158	0.000124	1.272289	0.636145
1e3	0.000616	0.000395	1.559382	0.779691
1e4	0.005412	0.005366	1.008610	0.504305
1e5	0.035431	0.034502	1.026932	0.513466
1e6	0.102981	0.121769	0.845712	0.422856
1e7	0.903584	0.530479	1.703335	0.851668
1e8	9.526111	4.937262	1.929432	0.964716
1e9	98.364100	52.274098	1.881699	0.940849

Tabela 2: Relação do tempo de execução, speedup e eficiência, de acordo com o tamanho da entrada, com seis casas de precisão. A opção escolhida na entrada do programa é *t* (somente imprimir o tempo na saída). Somente o tempo de execução do algoritmo em si é levado em conta. Note que para $N = 1e6$ tem-se um speedup menor que 1.

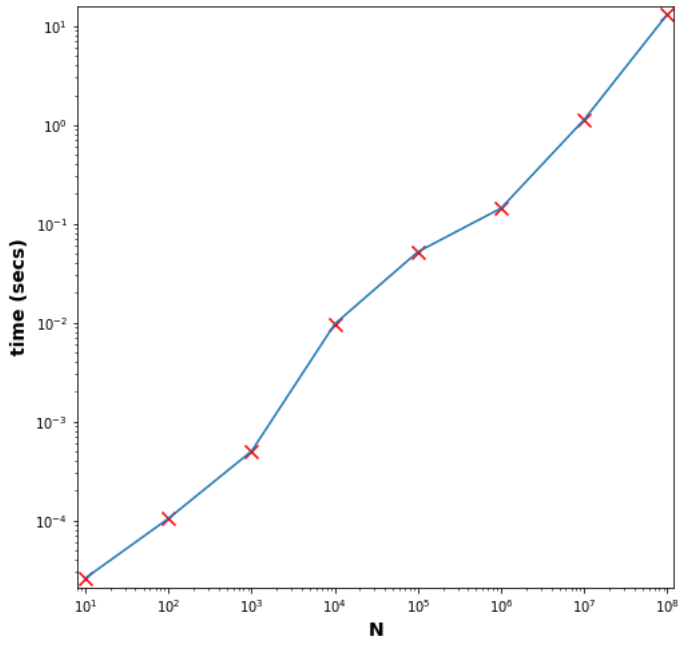

Figura 2: A curva mostra os resultados com a implementação utilizando listas para armazenar os números primos. É utilizada escala logarítmica, pois as entradas do problema estão separadas seguindo uma escala logarítmica.

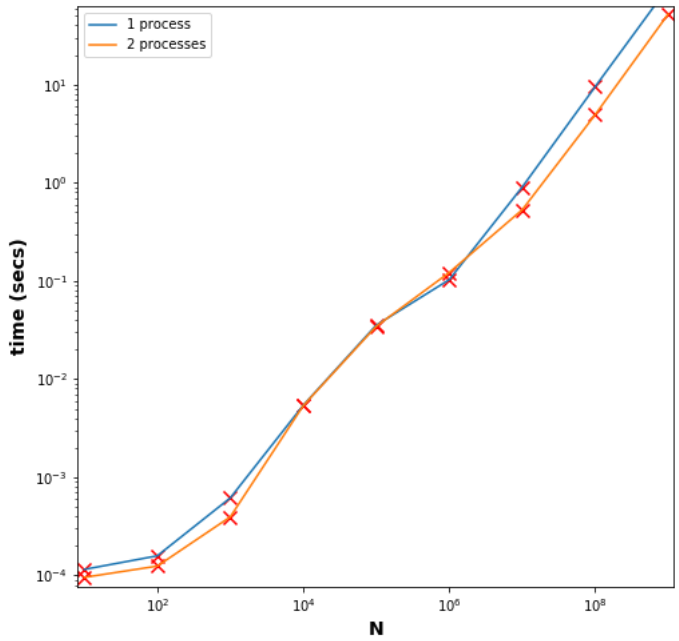

Figura 3: Comparação do tempo de execução para 1 e 2 processos, sob a implementação que utiliza um vetor. Os números são os mesmos vistos na Tabela 2.

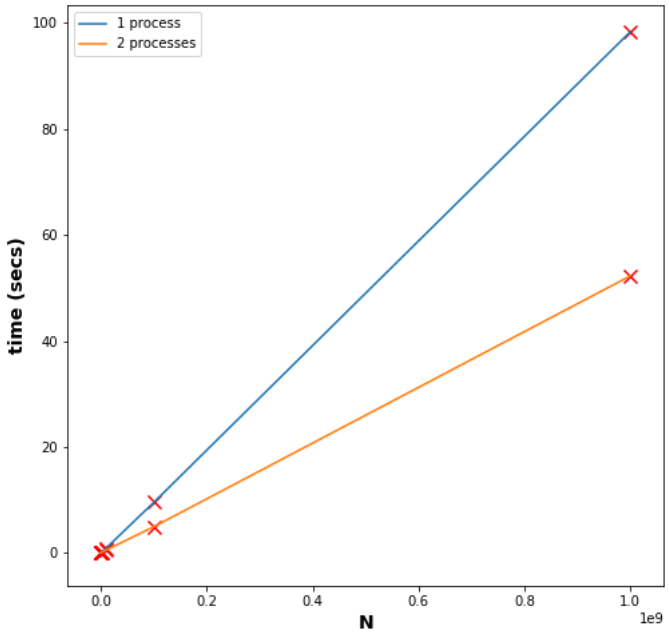

Figura 4: Comparação do tempo de execução, para 1 e 2 processos, sob a implementação que utiliza um vetor. Desta vez em escala linear para melhor ilustrar a diferença entre os dois.

4 Instruções de compilação

Para compilar o programa, basta digitar "make" no diretório principal. É necessário ter a biblioteca *boost* instalada em um diretório acessível globalmente por programas C/C++ (i.e. para habilitar *<boost/dynamic_bitset.hpp>*. Utilizamos o compilador GCC versão 9.3.0, e a versão de 2014 do padrão C++. A maioria dos compiladores deve produzir um programa correto, já que não utilizamos estruturas específicas do GCC. Para alterar o compilador a ser utilizado, ou outras configurações, basta modificar o arquivo *lib/make/global-vars-general.mk*. Para ativar otimizações do compilador, ou prover alguma bandeira específica para o compilador diretamente da interface de comando, basta digitar *make CL_OPS="<opcoes-de-linha-de-comando>"*. Um arquivo executável deve ser gerado na pasta *build*.

O script *testTimeBenchmarks* passa um conjunto de entradas para o programa, de acordo com os parâmetros fornecidos, e redireciona sua saída para arquivos de nomes únicos na pasta *tests*. O notebook *IPython* timePlots.ipynb provê um script que gera os visuais utilizados nesta apresentação. O repositório do projeto pode ser encontrado em [3].

Referências

[1] Open mpi. https://www.open-mpi.org/. Acessado em 1 de agosto de 2021.

[2] Boost library. https://www.boost.org/. Acessado em 1 de agosto de 2021.

[3] Repositório do projeto. https:https://github.com/Yowgf/eratosthenes-sieve. Acessado em 1 de agosto de 2021.