Relatório da Disciplina CC0021 - Programação Concorrente
Algoritmo: Crivo de Eratóstenes

Alunos: Cicero Samuel Santos Morais e Cicero José Ferreira Sousa

Agosto de 2021

1 Introdução

O Crivo de Eratóstenes é um algoritmo para encontrar os números primos em um intervalo [1, N]. No nosso projeto, fizemos três implementações deste algoritmo: sequencial; paralela com OpenMP; e paralela com MPI. Descreveremos em seguida as diferenças entre cada uma das implementações.

1.1 Sequencial

Esta é a versão "base" do algoritmo, sem nenhuma mudança ou adaptação. Nossa implementação segue os seguintes passos:

- Primeiro, criamos um vetor booleano de tamanho $N + 1$, abrangendo o intervalo [0, N]. O valor armazenado em cada índice do vetor equivale a primalidade do índice. Isto é, se o valor armazenado no índice i for true, então i é primo.
- Inicialmente, todos os valores do vetor são setados para true. Ou seja, assumimos que todos os números no intervalo são primos para começar.
- Setamos 0 e 1 como não-primos (false).
- Então, percorremos o vetor da seguinte maneira:

para p de 2 até $\sqrt{N}$, **faça:**
  se vetor[p] = true, **faça**:
    $i \leftarrow p^2$
    enquanto $i \leq N$, **faça**:
      vetor[i] $\leftarrow$ false
      $i \leftarrow i + p$
    fim do **para**
  fim do **se**
fim do **para**

- O algoritmo funciona da seguinte maneira: sabemos que o 2 é primo. Descartamos todos os seus múltiplos, pois obviamente eles não são primos. O próximo número não descartado que aparecer será primo. Eliminamos todos os seus múltiplos. Repetimos este processo até $\sqrt{N}$.
- Percorremos agora o vetor de 2 até N. Todos os números não descartados que sobraram são primos.

1.2 Paralelo com MP

Não há muita diferença entre esta implementação e a implementação sequencial. A única adaptação é que usamos a diretiva **#pragma omp for** para paralelizar os laços de para.

Na hora de percorrer o vetor e contar os primos, é usada a diretiva **#pragma omp for reduction(+ : count_primos)** para dividir a contagem entre as *threads*.

1.3 Paralelo com MPI

A implementação com MPI também segue a mesma lógica do algoritmo sequencial. Porém, aqui precisamos fazer alterações maiores. Nosso algoritmo do MPI segue a seguinte forma:

- Vamos dividir o vetor de primos entre os *cores* e cada um vai crivar sua parte seguindo o mesmo algoritmo mostrado na implementação sequencial.
- Para crivar todos os outros primos, precisamos apenas dos primos de 2 até $\sqrt{N}$. Portanto, vamos primeiro crivar o intervalo $[2, \sqrt{N}]$ e então enviar para cada *core* este intervalo crivado. Assim, cada *core* poderá crivar sua parte sem precisar enviar ou receber novas mensagens.
- O **rank 0** criva o intervalo $[2, \sqrt{N}]$ e envia para os outros *cores* através da função **MPI_Bcast**.
- Utilizando **MPI_Scatter** o vetor booleano é distribuído.
- Cada *core* vai crivar sua respectiva parte do vetor booleano e contar a quantidade de primos em sua parte.
- Por fim, o **rank 0** irá receber a quantidade de primos encontrada por cada *core* através da **MPI_Gather** e realizar a soma.

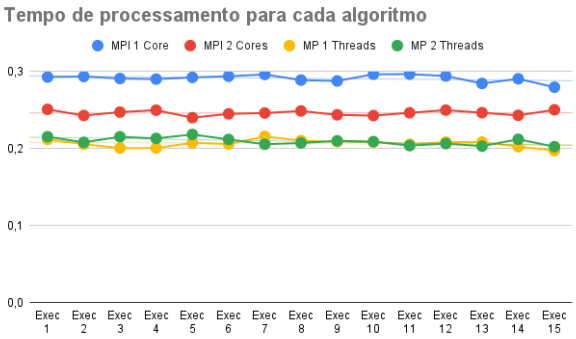
2 Resultados e Análise de Desempenho

Esta análise de desempenho foi realizada em uma máquina **Intel(R) Core(TM) i3-7200U @2.50 GHz com 2 núcleos** no **Ubuntu 19.04**.

Observação: as tabelas estão armazenadas em um arquivo do Google Sheets. Você pode verificar pelo link do rodapé.[1]

[1] Link para as tabelas da análise de desempenho.

Executamos cada um dos algoritmos 15 vezes com 10 milhões de elementos para serem crivados em situações similares e registramos as suas velocidades. Estes são os resultados obtidos:



2.1 Análise - Algoritmos MPI

A velocidade média do algoritmo MPI com 2 núcleos foi de 0.24s, com desvio padrão de 0.003. Já a do mesmo algoritmo com apenas 1 núcleo foi de 0,29s com desvio padrão de 0.004. Podemos notar que conseguimos um melhor desempenho no algoritmo MPI quando utilizamos os 2 núcleos ao invés de apenas 1 núcleo (o nosso "sequencial" para esta comparação). Neste caso, conseguimos um *SpeedUp* médio de 1.18 e uma eficiência média de 0.59.

O *SpeedUp* e, consequentemente, a eficiência, não foram muito altos quando comparamos o MPI 2 núcleos com o Sequencial (MPI 1 núcleo), porém foram significativos. Acreditamos que se aproveitássemos mais o potencial do MPI distribuindo o algoritmo ainda mais, estas métricas seriam melhores. Porém, não o fizemos pois não tínhamos acesso a um *cluster* e não tínhamos como conectar as máquinas dos integrantes em LAN devido ao isolamento. Desta forma, só pudemos testar o algoritmo MPI em uma única máquina, executando com 1 ou 2 núcleos.

2.2 Análise - Algoritmos MP

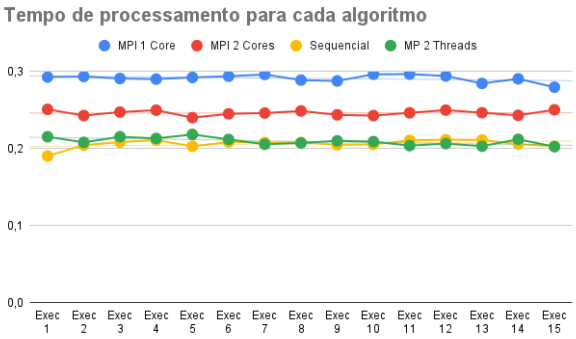
Apesar dos algoritmos em MP terem sido melhores que os do MPI em velocidade, podemos ver no gráfico que não houve muita diferença entre os dois algoritmos MP: com 2 *threads* e com 1 *thread* (o nosso sequencial para esta comparação). De fato, as médias foram, respectivamente, 0.209s e 0.206s, ambas com desvio de 0.004.

Acreditamos que a diferença de performance entre MP e MPI se deve ao custo de implementar paralelização com MPI ser mais elevado que o custo de implementar a paralelização com MP. Por causa disso, não podemos comparar MP com MPI, mas apenas comparar MP com MP.

Para entender melhor a falta de diferença entre os dois algoritmos em MP, resolvemos fazer alguns testes para responder a pergunta: **será que o algoritmo MP é paralelizável com baixa eficiência?**

2.2.1 Teste 1: O MP Sequencial (apenas 1 *thread*) tem algum custo com relação ao Sequencial simples?

Em vez de comparar o MP 2 *threads* com o MP 1 *thread*, comparamos o MP 2 *threads* com o algoritmo Sequencial simples. Isto porque queríamos saber se havia algum custo de desempenho apenas para implementar o MP com 1 *thread*, mesmo que ele fosse praticamente idêntico ao Sequencial simples. Estes são os resultados:



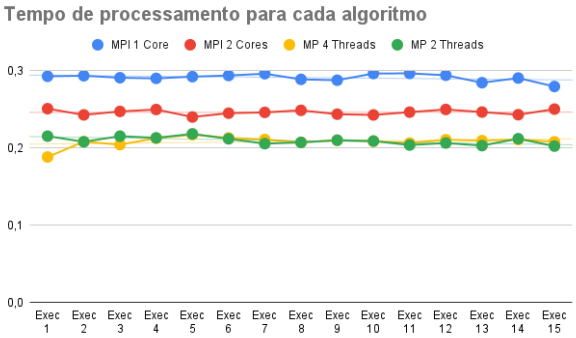
Novamente, tivemos uma velocidade média de 0.209s para o MP 2 *threads* e de 0.206s para o Sequencial simples, sendo que o primeiro com desvio padrão de 0.004 e o último com desvio de 0.005. Concluímos que não há um custo significativo em trocar o Sequencial simples por um Sequencial com MP.

2.2.2 Teste 2: Aumentar o número de *threads* melhora o desempenho?

Em vez de rodar um Sequencial com MP (1 *thread*, rodamos dois paralelos com MP: um com 2 *threads* e um com 4 *threads*. Queríamos descobrir se aumentando a quantidade de unidades de processamento, obteríamos um desempenho melhor. Eis os resultados:

E vemos que, mais uma vez, não obtivemos mudanças significativas. Tivemos um média de 0.209s para o MP 2 *threads* e de 0.208s para o MP 4 *threads*, com desvios padrão de 0.004 e 0.006 respectivamente.

Na verdade, nem aumentando a quantidade de elementos a serem crivados, conseguimos obter mudanças significativas entre as implementações com MP. Tivemos um *SpeedUp* médio de 0.986 com desvio padrão de 0.03. Nossa eficácia para o MP foi de 0.49, com desvio padrão de 0.016.



3 Conclusões

Com base no que vimos na análise, concluímos que devemos obter um melhor desempenho paralelizando o crivo com MPI. Já conseguimos verificar essa melhora de desempenho distribuindo entre 2 núcleos, apesar de ser uma melhora relativamente pequena devido às limitações que temos em nossas máquinas. Porém, achamos que é provável que consigamos melhorar nossas métricas aumentando a distribuição entre mais núcleos e dispositivos.

Já no algoritmo paralelizado em MP, não obtivemos resultados tão bons. Chegamos a um $SpeedUp < 1$ e uma eficiência bem baixa em diferentes testes. Estes vários testes que fizemos nos leva assim a afirmar que **o algoritmo é paralelizável com baixa eficiência em MP**.