

Projeto de Programação 1

Crivo de Eratóstenes usando MPI

por Elsie Antunes Junior [11097612] <eantunes@aluno.ufabc.edu.br>

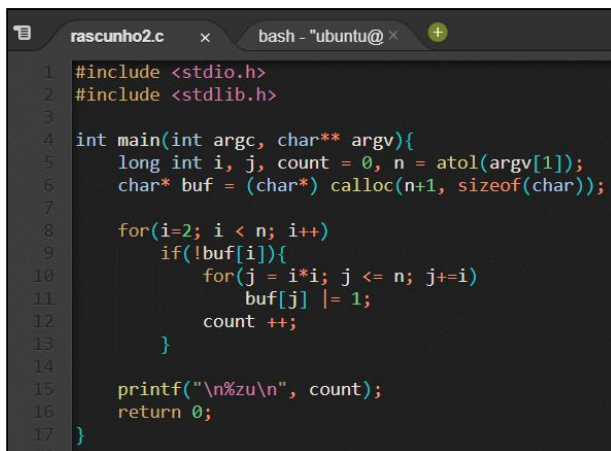
MCZA020-13 - Programação Paralela 2019.Q1

Professor: [Emilio Franceschini](#)

Introdução

O clássico Crivo de Eratóstenes (276 A.C. - 194 A.C.) é um algoritmo para a obtenção de números primos menores que um determinado número dado. Basicamente, o processo consiste em eliminar os múltiplos a frente de cada primo encontrado. Para paralelizar o trabalho é necessário tornar o algoritmo paralelizável. Ele não é em todo, pois percorre uma lista várias vezes, causando dependências, no mínimo de seus ponteiros. A possibilidade existente de se desmembrar as dependências tornam o código algo digno de estudos em nível acadêmico. A utilização de *wheels* (cuja tradução, "rodas", na minha opinião, não nos remete ao que é feito de fato) é a forma mais interessante de obter melhores resultados deste algoritmo e quiçá de descobrir novas propriedades dos números primos que, fora de um contexto acadêmico, ficam tão nebulosos nas árduas leituras disponíveis. Como tudo que é rico e interessante, o enunciado é simples mas a implementação na prática enche o código de abstrações. O bom programador se perde, o mau nem tenta. Na minha opinião, se faz necessário que mais evangelistas matemáticos e computeiros surjam para elucidar esses mistérios atualmente só acessíveis aos que vivem de estudo.

O Algoritmo



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char** argv){
5     long int i, j, count = 0, n = atol(argv[1]);
6     char* buf = (char*) calloc(n+1, sizeof(char));
7
8     for(i=2; i < n; i++)
9         if(!buf[i]){
10             for(j = i*i; j <= n; j+=i)
11                 buf[j] |= 1;
12             count ++;
13         }
14
15     printf("\n%zu\n", count);
16     return 0;
17 }
```

O Crivo encontra todos os números primos menores que um determinado valor n . Partindo de uma lista de naturais (aqui representada pelos índices do vetor alocado com tamanho n), e utilizando-se de dois loops, consulta-se cada um dos campos, a partir do segundo, que representa o número 2, e marca-se todos os múltiplos desse número, a frente na lista. Repete-se o loop interno para todos os números não marcados. Ao final, todos os campos não marcados representarão a

lista de números primos. O exemplo mostrado apenas conta as ocorrências. Para exibí-los bastaria imprimir o valor do índice a partir do escopo do *if* na linha 9 (geralmente, mas não necessariamente, na linha 12).

Complexidade

Esta versão do crivo tem dois laços aninhados. O primeiro percorre toda a lista (este não precisaria) consultando se executa o próximo loop, portanto $O(n)$. O segundo loop, para cada fator primo i , percorre a lista de i^2 até n ao passo i . Ou seja, para cada fator primo temos

$$\frac{n - i^2 + 1}{i} = O(n)$$

Esse loop é executado para cada fator primo encontrado. A quantidade de fatores primos da lista é dada pela função $\pi(x)$ conjecturada por Gauss e Legendre.

Segundo a teoria do número aproximadamente constante, $\pi(x) = \Theta\left(\frac{x}{\log x}\right)$ primo, a razão $x/\pi(x)$ é mesmo para valores grandes de x . O que sugere que uma boa aproximação para $\pi(x)$ é:

$$\pi(x) = O(\log x)$$

Assim sendo, o segundo laço é executado, aproximadamente

$$\pi(\sqrt{n}) = O(\log \sqrt{n})$$

vezes. Utilizando a aproximação $\sqrt{n} = O(\log n)$, tem-se que o segundo laço será executado $\pi(\log n) = O(\log \log n)$ vezes. Como a complexidade para cada fator primo é $O(n)$, a complexidade do segundo laço e, portanto, do algoritmo é:

$$n \cdot O(\log \log n) = O(n \cdot \log \log n)$$

A complexidade de espaço nesta versão primitiva é dada pelo tamanho da lista que armazena o estado de cada número (primo ou não). Portanto, $O(n)$.

Otimizações

Como todo algoritmo, entre o caso geral e o concreto, específico, otimizações podem ser implementadas. O crivo original não previa, por exemplo, que as marcações poderiam iniciar no quadrado de cada primo encontrado. Esta é uma observação matemática de que um primo igual ou menor que a raiz é suficiente para fatorá-lo (se fatorável for) qualquer inteiro, não é atribuída necessariamente a Eratóstenes. Uma otimização menor, porém significativa computacionalmente, é a utilização de uma estrutura de dados que possibilite a registrar o estado do número em bits. A busca, a marcação, a contagem e a representação dos números podem ser beneficiadas, sendo revertidas em economia de espaço e aumento de velocidade devido a um melhor uso do *cache*.

As duas principais otimizações do crivo são a segmentação e a utilização de *wheels* (cuja tradução, "rodas", na minha opinião, não nos remete ao que é feito de fato). A segmentação parte de uma ideia simples, de divisão da lista em pedaços menores que faz muito sentido computacionalmente, pois existe a possibilidade de execução paralela de várias máquinas (praticamente) ao mesmo tempo. *Wheels* é provavelmente o fato matemático que melhor

ajuda na busca por números primos. Como sabem, só existe um número primo par. Faz sentido, portanto, construir uma estrutura de dados que ignore os números pares a partir do 2. Pelo mesmo princípio, o único número primo múltiplo de 3 é apenas o próprio 3. A estrutura pode, pois, ignorar os espaços dos múltiplos de 3 após o segundo primo. Seguindo este princípio (que vai ao infinito) temos classes de “*podas*” no crivo que, se por um lado diminuem drasticamente o processamento e o espaço necessário para a obtenção de primos cada vez maiores, aumentam significativamente a complexidade do código.

Segmentação

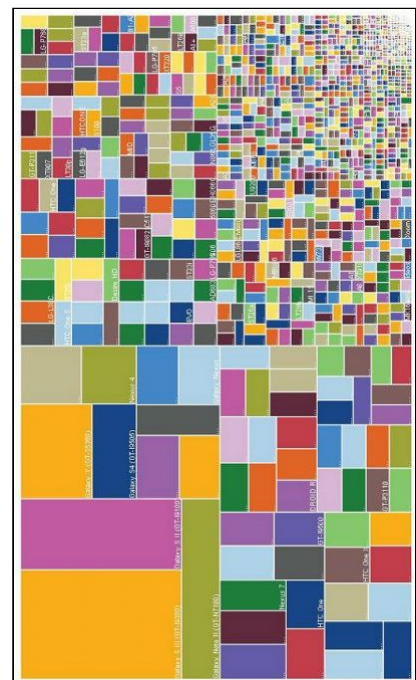
Para paralelizar o trabalho é necessário tornar o algoritmo paralelizável. Ele não é em todo, pois percorre uma lista várias vezes, causando dependências, no mínimo de seus ponteiros. A possibilidade existente de se desmembrar as dependências tornam o código algo digno de estudos em nível acadêmico. Foram encontradas pelo menos três soluções para dividir o trabalho. Uma delas consiste em dividir o espaço, calcular todos os primos menores que a raiz quadrada de n ligado a um sistema de mensagens que distribui os primos para cada processo marcar o seu segmento. Outra estratégia seria de divisão de tarefa. Todos os processos atuando sobre uma lista compartilhada e cada um marca uma faixa de primos. A abordagem deste trabalho eu concentrei na divisão do espaço do total de números com a iteração de cada primo agindo sobre um segmento.

Para começar, antes da utilização do MPI para a paralelização propriamente dita, construí uma versão (foram várias, na verdade) intermediária que já executa a divisão de n em p intervalos, onde p é o número de processos, resultando em segmentos de tamanho **seg**. Em uma próxima implementação, fica registrado, será possível implementar uma divisão ponderada, proporcional aos resultados de marcação de tempo. O tratamento do resto da divisão foi feito de forma que o último processo fique um segmento menor.

Posto isto, é chamada a função `trabalho()` que recebe como parâmetros o buffer previamente preparado, o n e um intervalo `[low, high]` e retorna o número de marcações que **não** fez neste buffer. Inicialmente o buffer ficava encapsulado neste método. Mas foi retirado de lá para facilitar o retorno da lista de vinte primos pedida no enunciado.

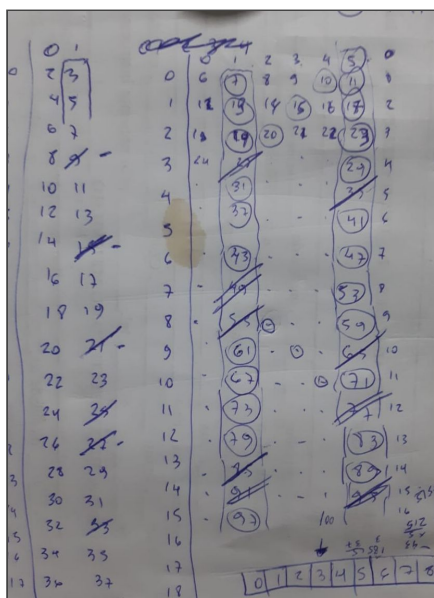
A função função `trabalho()` percorre uma lista de tamanho \sqrt{n} marcando primos no seu próprio buffer e ‘enviando’ cada primo encontrado para a função `markSegBuf()` proceder a marcação de um segmento de intervalo `[low, high]`. Ao final, as contagens recebidas de cada iteração são enviadas para a `main()` fazer a exibição adequada.

Por fim, as funções `markSqrBuf()` e `markSegBuf()` são o coração do algoritmo. Deveriam ser essencialmente iguais pois fazem basicamente a mesma coisa. Fica registrado mais uma meta para o próximo trabalho, unificá-las. O funcionamento delas é complexo então dedico lhes uma sessão inteira.



Wheels

A utilização de *wheels* (cuja tradução, "rodas", na minha opinião, não nos remete ao que é feito de fato) é a forma mais interessante de obter melhores resultados deste algoritmo e quiçá de descobrir novas propriedades dos números primos que, fora de um contexto acadêmico, ficam tão nebulosos nas árduas leituras disponíveis. Como tudo que é rico e interessante, o enunciado é simples mas a implementação na prática enche o código de abstrações. O bom programador se perde, o mau nem tenta. Na minha opinião, se faz necessário que mais evangelistas matemáticos e computadores surjam para elucidar esses mistérios atualmente só acessíveis aos que vivem de estudo.



The image shows a handwritten 'wheels' diagram, which is a sieve for finding primes. It consists of a grid of numbers arranged in columns. The first column contains numbers from 0 to 19. The second column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The third column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The fourth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The fifth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The sixth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The seventh column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The eighth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The ninth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The tenth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The eleventh column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The twelfth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The thirteenth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The fourteenth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The fifteenth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The sixteenth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The seventeenth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The eighteenth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The nineteenth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18). The twentieth column contains numbers from 0 to 19, with some numbers circled (2, 3, 5, 7, 11, 13, 17, 19) and some crossed out (4, 6, 8, 10, 12, 14, 16, 18).

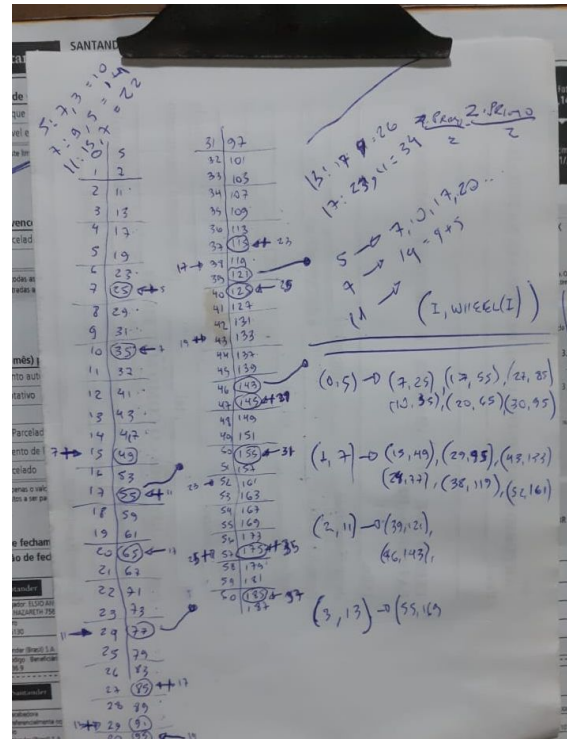
Como sabem, só existe um número primo par. Faz sentido, portanto, construir uma estrutura de dados que ignore os números pares a partir do 2. Pelo mesmo princípio, o único número primo múltiplo de 3 é apenas o próprio 3. A estrutura pode, pois, ignorar os espaços dos múltiplos de 3 após o segundo primo. Seguindo este princípio (que vai ao infinito) temos classes de "podas" no crivo que, se por um lado diminuem drasticamente o processamento e o espaço necessário para a obtenção de primos cada vez maiores, aumentam significativamente a complexidade do código. A figura ao lado, que eu passei o último mês olhando no ônibus a caminho do serviço e na hora do almoço, mostra que se você alinhar os números em duas colunas verá que os pares ficam de um lado e não precisarão nem ser riscados pois seriam todos. Se você alinhá-los em $2 \times 3 = 6$

colunas poderá trabalhar com apenas duas delas. O mesmo aconteceria com $2 \times 3 \times 5 = 30$ colunas ou $2 \times 3 \times 7 = 210$ colunas. Sempre que a montagem for do mesmo número de colunas que o produto dos n primeiros primos isso vai acontecer. O presente trabalho concentrou-se em 2 de 6 colunas. Fica registrado a meta para implementar tantas colunas quantas forem necessárias para infinitos números. Quando lidamos com essas duas colunas novas propriedades surgem. Por exemplo, encontre o primeiro primo no topo da coluna $6k+5$, por exemplo, o número 5. Agora conte 5 linhas para baixo e risque, mais 5 e risque, etc, etc. Para cada primo P encontrado, metade dos múltiplos dele se encontram a cada P linhas para baixo. Ora, mas só é necessário riscar os números a partir de P^2 . Por exemplo, ao encontrar o primo 11, se você contar a 11-ésima linha não precisará riscá-la pois ela já terá sido riscada por um primo menor que 11. Manualmente é até divertido fazer a operacionalização. Mas em código, como fica?

Todo o trabalho das funções `markSqrBuf()` e `markSegBuf()` é justamente de encontrar um cálculo simples que nos remeta às coordenadas de P^2 para então marcar os seus sucessores. Foram seguidas duas estratégias diferentes pois matemática não é exatamente o meu forte e as leituras a respeito são extremamente árduas.

Para começar `markSqrBuf()` que recebe um primo e precisa alinhar seus múltiplos em uma sequência para um `loop for` operacionalizar a marcação. A estratégia aqui foi

transformar as duas colunas $6k+1$ e $6k-1$ em uma única coluna contendo todos estes candidatos a primos $\{5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37 \dots\}$ $x+2, +4, +2, +4 \dots$ } e observar que ganha-se uma nova propriedade: metade dos valores que precisam ser riscados estão a uma distância $2P$ de cada primo e a outra metade está a uma distância $2P$ do quadrado deste número. Ora, basta encontrar o primo P , contar $2P$ índices da sequência e marcar o próximo, depois encontrar o índice de P^2 , contar $2P$ e marcar o próximo. No entanto, havia uma dificuldade: como encontrar o índice de P^2 ? Novamente fiz um desenho e carreguei ele comigo por todo lugar que eu fosse. Eu tinha os números de entrada e os números de saída e precisava encontrar uma função que ligasse uma coisa a outra. A primeira coisa que fiz foram os macros `wheel()` e `antiwheel()` cuja função é converter índice



no seu candidato a primo e vice-versa. A próxima coisa a se notar é que se duas sequências periódicas de mesmo espaçamento convivem juntas, a distância entre elas é menor que o período. Em outras palavras, a cada $2P$ índices desse conjunto (lembrando que o conjunto não tem todos os múltiplos de P , apenas os do tipo $6k+1$ e $6k-1$) eu tenho um múltiplo de P , mas antes de chegar a ele eu passo por outro múltiplo de P , vindo da sequência que começa pelo P^2 . Para encurtar a história, conjecturei (baseado em observação que não consigo provar formalmente) que no segundo terço de cada intervalo entre P e $2P$ eu encontro com o outro múltiplo de P . Então fiz uma função `next()` (na verdade, um macro) que, dado o índice do candidato a primo que encontrei sequencialmente me devolva o índice daquela sequência que, teoricamente, contém o P^2 . Eu estava errado? em partes. A estratégia me permite, de fato, com dois loops marcar todos os múltiplos de P , e assim eu o fiz, como mostra a imagem.

```

8
9 // i: índice do primo vindo da função trabalho()
10 // prime: candidato a primo do tipo 6k+1 U 6k-1
11 // set(j): marca no buffer que j é composto
12
13 prime = wheel(i);
14
15 j = i + 2*prime;
16 for(; wheel(j) < sqr; j += 2*prime)
17     set(j);
18
19 j = i + 2*prime - next(prime);
20 for(; wheel(j) < sqr; j += 2*prime)
21     set(j);
22

```

O que poderá ser visto na implementação é algo equivalente, fruto de *escovação de bits* onde eu procurei transformar esses dois loops em um único. Talvez não devesse te-lo feito. Fica registrado para futuras implementações. Sem querer me alongar, continuo agora com a estratégia usada na função `markSegBuf()`, que, como já disse, é essencialmente a mesma função, mas que eu usei outra estratégia.

O que função `markSegBuf()` precisava fazer, assim como esta última, era receber um primo e encontrar uma forma de marcar os múltiplos no buffer economizando o máximo de processamento e espaço possível, com um adicional: agora a função recebe um primo e um intervalo `[low, high]` que delimita o segmento para ser marcado com este primo. Ou seja, o agente neste ponto precisa riscar números compostos em um intervalo de inteiros sem sequer saber qual o tamanho do resto do trabalho, de forma totalmente independente. E para isso precisaria calcular em que *offset* do intervalo o primeiro primo deveria ser marcado. Igualmente a propriedade da distância entre metade dos múltiplos ser de $2P$ e a outra metade também, só que num *offset* baseado na posição de P^2 .

Para encontrar o índice da primeira coluna, a que contém P^2 é simples com a utilização do macro `antiwheel(prime*prime)` nos remete diretamente à posição de P^2 . Matematicamente eu deveria saber transportar este índice para dentro do intervalo `[low, high]` para marcar no buffer quais números não são primos mas eu falhei em conseguir uma função baseada em restos de divisão e preferi

```
// o primeiro loop poderia ser substituido
// por um cálculo analítico?
first = antiwheel(prime*prime);

while(first < low) // vergonha
    first += 2*prime;

for(i = first; i <= high; i += 2*prime)
    set(i-x);
```

fazer um loop que incrementa a variável `first` até obter o primeiro múltiplo de $2P$ maior que `low` e assim marcar até `high` de $2P$ em $2P$. Fica registrado aqui um dos gargalos que podem ser atacados para aumento significativo de performance.

Eu sei, você pode estar se perguntando: “se era tão simples encontrar o P^2 então porque não fez isso na outra função?” Agora é só andar $2P$ e voltar um terço e chegará na outra coluna certo? não. Minha suposição de que dado P bastaria avançar $4P/3+1$ (ou voltar $\text{floor}(2P/3)$ que é o que dá certo) para encontrar o outro, como um relógio parado - que está certo duas vezes por dia - carece de observações. Quando eu percorro o vetor e encontro um primo não marcado ele pode ser do tipo $6k+1$ ou $6k-1$. Com um pouco mais de matemática é possível provar que os quadrados dos números estarão sempre na coluna $6k+1$ e nunca na coluna $6k-1$. Quando eu pegava um primo P de qualquer coluna e somava $2P$ e voltava $2P/3$ no conjunto $\{6k+1 \cup 6k-1\}$ eu chegava de fato no múltiplo de P , só que ele poderia ser de qualquer das duas colunas. Aqui eu estou marcando primeiro os da coluna $6k+1$ e nela eu consigo encontrar o índice dos quadrados dos primos, na outra não tem. E se eu for seguir a estratégia anterior vou conseguir o outro índice só em metade das vezes. Então, novamente, enumerei os números que eu tenho de um lado e os números que preciso chegar do outro e procurei uma relação. A sequência $\{1, 4, 3, 8, 5, 12, 7, 9, \dots\}$ inicialmente não fazia sentido. Procurei no Oeis que me indicava uma sequência parecida mas que não encaixava $\{1, 4, 3, 8, 5, 12, 7, (16), 9, \dots\}$ e então me dei conta! a minha

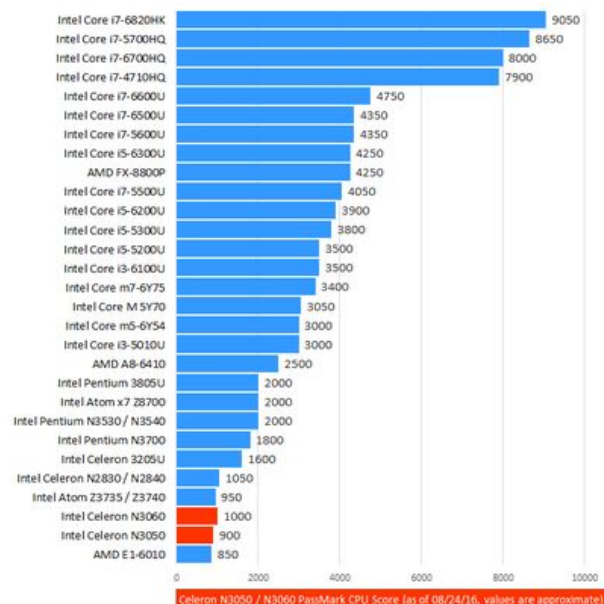
sequência foi baseada em primos, mas deveria ter sido baseada, de forma genérica, em candidatos à primos! Depois de refletir a respeito cheguei a conclusão de que o que está sendo feito nas duas funções `markSqrBuf()` e `markSegBuf()` é exatamente a mesma coisa. Existe mesmo uma relação de 2P/3 entre os **candidatos** à primo que estão posicionados no formato $x+2$, $+4$, $+2$, $+4$, $+2$...

Infelizmente, não há mais tempo para elaborar um código mais correto. Outras disciplinas, meu serviço e meus filhos também precisam de atenção (e como precisam).

Dados técnicos

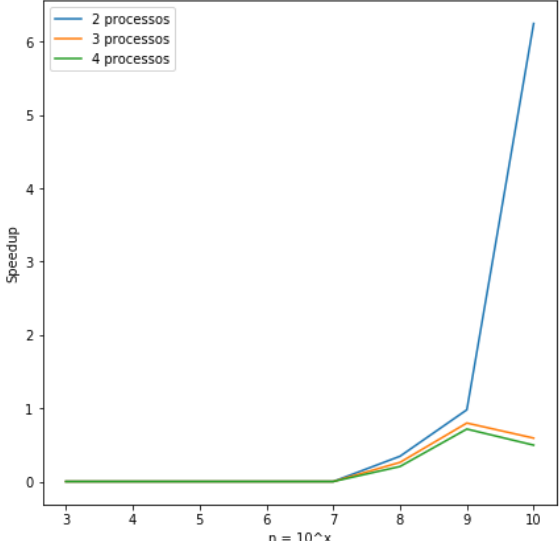
A composição do código foi feita em um Chromebook Samsung, Intel Celeron N3050 - 1.60GHz com 2 CPUs (1 thread por core) 24k de cache L1D e 2GB de RAM. Alguns testes foram feitos na plataforma online Cloud9 - www.c9.io cuja especificação disponível aponta para um i5-3570 @ 3.40GHz com 4 CPUs (1 tpc), L1D 32k e 512MB de RAM.

A indisponibilidade de um *'supercomputador'* não é impedimento para a aplicação da maior parte dos fundamentos. As dificuldades são comparáveis a outras no campo social, que impactam de forma muito mais significativa.



Gráficos de Speedup

Infelizmente, devido a inconstância das máquinas utilizadas não houve possibilidade de medir o tempo de forma adequada. O serviço online parece ter um sistema de punição para quem roda várias vezes um programa que consome processamento acima de uma média, causando overheads que estragam os dados. O que consegui no Chromebook também não parecia adequado. Em todos os testes, infelizmente, o serial sempre obteve melhores tempos que o serial. Talvez quando o professor testar em suas supermáquinas a real vantagem teórica fique melhor explicitada.

<p>Com 2 processos, cloud9</p> <p>10e3 $(0.7 + 0.6 + 0.7)/3 = 0.7$ 10e4 $(0.7 + 0.7 + 0.7)/3 = 0.7$ 10e5 $(0.7 + 0.7 + 0.7)/3 = 0.7$ 10e6 $(0.7 + 0.7 + 0.7)/3 = 0.7$ 10e7 $(0.7 + 0.7 + 0.7)/3 = 0.7$ 10e8 $(1.2 + 1.2 + 1.1)/3 = 1.16$ 10e9 $(10.8 + 10.6 + 11.3)/3 = 10.9$ 10e10 $(164.6 + 163.0 + 162.5)/3 = 16.34$ 10e11 killed!</p>	<p>Com 3 processos, cloud9</p> <p>10e3 $(1.0 + 0.9 + 1.0)/3 = 0.97$ 10e4 $(1.1 + 1.0 + 1.0)/3 = 1.04$ 10e5 $(1.2 + 1.0 + 1.0)/3 = 1.07$ 10e6 $(1.0 + 1.0 + 1.0)/3 = 1.00$ 10e7 $(1.0 + 1.1 + 1.3)/3 = 1.13$ 10e8 $(1.6 + 1.6 + 1.5)/3 = 1.57$ 10e9 $(13.7 + 13.0 + 13.4)/3 = 13.37$ 10e10 $(171.8 + 179.2 + 166.1)/3 = 172.37$ 10e11 killed!</p>																																				
<p>serial, cloud9</p> <p>10e3 $(0.0 + 0.0 + 0.0)/3 = 0.00$ 10e4 $(0.0 + 0.0 + 0.0)/3 = 0.00$ 10e5 $(0.0 + 0.0 + 0.0)/3 = 0.00$ 10e6 $(0.0 + 0.0 + 0.0)/3 = 0.00$ 10e7 $(0.0 + 0.0 + 0.0)/3 = 0.00$ 10e8 $(0.4 + 0.4 + 0.4)/3 = 0.40$ 10e9 $(11.1 + 10.3 + 10.6)/3 = 10.67$ 10e10 $(143.9 + 147.3 + 15.2)/3 = 102.13$ 10e11 killed!</p>	 <table><caption>Approximate data points from the Speedup graph</caption><thead><tr><th>n = 10^x</th><th>2 processos</th><th>3 processos</th><th>4 processos</th></tr></thead><tbody><tr><td>3</td><td>0.00</td><td>0.00</td><td>0.00</td></tr><tr><td>4</td><td>0.00</td><td>0.00</td><td>0.00</td></tr><tr><td>5</td><td>0.00</td><td>0.00</td><td>0.00</td></tr><tr><td>6</td><td>0.00</td><td>0.00</td><td>0.00</td></tr><tr><td>7</td><td>0.00</td><td>0.00</td><td>0.00</td></tr><tr><td>8</td><td>0.40</td><td>0.30</td><td>0.20</td></tr><tr><td>9</td><td>1.07</td><td>0.80</td><td>0.60</td></tr><tr><td>10</td><td>6.34</td><td>0.70</td><td>0.50</td></tr></tbody></table>	n = 10 ^x	2 processos	3 processos	4 processos	3	0.00	0.00	0.00	4	0.00	0.00	0.00	5	0.00	0.00	0.00	6	0.00	0.00	0.00	7	0.00	0.00	0.00	8	0.40	0.30	0.20	9	1.07	0.80	0.60	10	6.34	0.70	0.50
n = 10 ^x	2 processos	3 processos	4 processos																																		
3	0.00	0.00	0.00																																		
4	0.00	0.00	0.00																																		
5	0.00	0.00	0.00																																		
6	0.00	0.00	0.00																																		
7	0.00	0.00	0.00																																		
8	0.40	0.30	0.20																																		
9	1.07	0.80	0.60																																		
10	6.34	0.70	0.50																																		
<p>3 testes com 4 processos rodando com n = 9</p> <p>Máquina 0 rodou em $(4.835+5.598+5.094)/3 = 5.175$ Máquina 1 rodou em $(6.979+7.572+6.732)/3 = 7.094$ Máquina 2 rodou em $(8.525+8.403+7.984)/3 = 8.304$ Máquina 3 rodou em $(9.418+9.367+9.913)/3 = 9.566$</p> <p>A distribuição de tarefa não foi ponderada no código. A natureza do trabalho sugere que devemos privilegiar mais os últimos processos, não por terem maior quantidade de primos (são aproximadamente equivalentes entre os segmentos) mas por serem números maiores fazendo com que o trabalho de busca pelo índice quadrado do candidato a primo e a posterior trazida dele ao intervalo do segmento, seja excessivo devido a escolha de calculá-lo via loop. Questão esta que será explicitada numa próxima oportunidade.</p>																																					

Bibliografia

O'Neill, Melissa E., "[The Genuine Sieve of Eratosthenes](#)", *Journal of Functional Programming*, published online by Cambridge University Press 9 October 2008 [doi:10.1017/S0956796808007004](#), pp. 10, 11 (contains two incremental sieves in Haskell: a priority-queue-based one by O'Neill and a list-based, by Richard Bird).

[Jonathan Sorenson, *An Introduction to Prime Number Sieves*](#), Computer Sciences Technical Report #909, Department of Computer Sciences University of Wisconsin-Madison, January 2, 1990 (the use of optimization of starting from squares, and thus using only the numbers whose square is below the upper limit, is shown).

Além de definições e artigos disponíveis na internet e outros trabalhos que citam Paul Pritchard, Sorenson e outros autores interessados na computabilidade do assunto. São livros densos e em inglês, confesso que tentei absorver o máximo que pude. Adoraria ter condições de dedicar mais tempo para o assunto.