Mestrado Integrado em Engenharia
Informática e Computação

Computação Paralela

Sieve of Eratosthenes

Implementação e Análise de Desempenho

João Henrique Poceiro Vieira de Araújo - 201303962
Rúben José da Silva Torres - 201405612

19 May 2019

Conteúdo

1 Introdução

No âmbito da unidade curricular de Computação Paralela, foi proposta a realização de um estudo sobre a paralelização de um algoritmo, através do uso da API **OpenMP** para paralelização com memória partilhada e a biblioteca **OpenMPI** para a paralelização com memória distribuída. O algoritmo escolhido para implementação e análise foi o *Sieve of Eratosthenes*.

O *Sieve of Eratosthenes* é um algoritmo simples para encontrar todos os números primos num intervalo $[2, n] : n \in N \backslash \{1\}$. Um número é considerado primo se e só se for divisível por ele próprio e por 1. O objetivo principal deste problema será paralelizar o algoritmo de forma a obter melhor desempenho e escalabilidade do que a sua versão sequencial. Para este efeito foram implementadas 4 versões do algoritmo:

- Algoritmo 1 - Versão sequencial
- Algoritmo 2 - Versão com memória partilhada
- Algoritmo 3 - Versão com memória distribuída
- Algoritmo 4 - Versão híbrida com memória partilhada e distribuída

Este relatório tem como objetivo discutir as abordagens à implementação deste algoritmo, efetuar uma análise e comparação aos resultados obtidos, e tirar as conclusões apropriadas.

2 Descrição do Problema

O algoritmo *Sieve of Eratosthenes* atua sobre uma lista de n números inteiros, marcando iterativamente todos os múltiplos dos números primos inferiores ou iguais a $\sqrt{n}$.

Figura 1: Exemplo do *Sieve of Eratosthenes*

No final da execução do algoritmo, todos os números que não se encontram marcados representam o conjunto de números primos encontrados nesse intervalo.

A complexidade temporal e espacial deste algoritmo é Θ (n ln ln n) e Θ (n), respetivamente. Sendo a sua implementação em pseudo-código a seguinte:

```
Input: an integer n > 1.

Let A be an array of Boolean values, indexed by integers 2 to n,
initially all set to true.

for i = 2, 3, 4, ..., not exceeding √n:
  if A[i] is true:
    for j = i², i²+i, i²+2i, i²+3i, ..., not exceeding n:
      A[j] := false.

Output: all i such that A[i] is true.
```

Figura 2: Pseudo-código do *Sieve of Eratosthenes*

3 Solução Sequencial - Algoritmo 1

Para a implementação **sequencial** do algoritmo, foi otimizado o pseudo-código descrito na secção anterior modo a que este não tenha em conta os números pares aumentando a velocidade de processamento.

```
1   n = pow(2,n);
2   unsigned long long k = 3;
3   do
4   {
5       for (unsigned long long j = k*k ; j<n ; j+=2*k)
6       {
7           primes[j>>1]=true;
8       }
9       do
10      {
11          k+=2;
12      } while (k*k <= n && primes[k>>1]);
13  } while (k*k <= n);
```

3.1 Tempo de execução do algoritmo

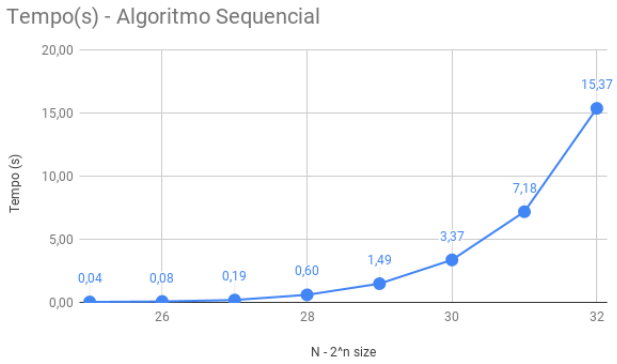

Figura 3: Tempo de execução do algoritmo 1

Pode concluir-se que o tempo decorrido aumenta em consoante o aumento do n. Verifica-se ainda que esse aumento é próximo do fator 2, o que corresponde também ao aumento da dimensão do intervalo de primos a serem gerados.

4 Soluções Paralelas

De forma a melhorar o desempenho do algoritmo, é possível paralelizar partes da implementação sequencial. Este paralelismo pode ser feito de maneiras distintas, quer seja partilhando a memória de um processo em execução por várias *threads*, ou distribuindo blocos de memória por vários processos. É possível ainda fazer uso de ambas estas soluções, obtendo-se um paralelismo híbrido.

4.1 Versão com memória partilhada

A implementação com memória partilhada utiliza a API **OpenMP** para atribuir a *threads* diferentes parte da marcação dos números múltiplos, aquando da descoberta de um número primo. A alteração feita ao código do algoritmo sequencial encontra-se no seguinte excerto:

```
1        #pragma omp parallel for num_threads(n_threads)
2        for (unsigned long long j = k*k ; j<n ; j+=2*k)
3        {
4            primes[j>>1]=true;
5        }
```

A utilização da diretiva `#pragma omp parallel for` permite que o ciclo seja executado concorrentemente pelos vários *threads* do processo.

4.2 Versão com memória distribuída

No caso da implementação com memória distribuída, é utilizada a biblioteca **OpenMPI** para dividir a lista de números pelos processos que são executados. Cada processo tem um bloco de memória local associado, no qual é feita a marcação dos múltiplos de um número primo, enviado pelo processo principal (rank = 0) através de *Broadcast*. Esta troca de mensagens é feita após cada ciclo de cálculo, altura em que o processo principal descobre o próximo número primo a calcular e volta a transmiti-lo.

O código referente a esta implementação pode ser encontrado no ficheiro ***openMPI.cpp*** e as alterações feitas são as seguintes:

- Calcular o tamanho do bloco de memória local associado a cada processo, e o primeiro e último índices desse mesmo bloco;

```
1    #define BLOCK_LOW(i, n, k) ((i) * (n) / (k))
2    #define BLOCK_HIGH(i, n, k) (BLOCK_LOW((i) + 1, n, k) - 1)
3    #define BLOCK_SIZE(i, n, k) (BLOCK_LOW((i) + 1, n, k) - BLOCK_LOW(i, n, k))
4
5    unsigned long long blockSize = BLOCK_SIZE(worldRank, n-1, worldSize);
```

```
6     unsigned long long blockLow = 2+BLOCK_LOW(worldRank, n-1, worldSize);
7     unsigned long long blockHigh = 2+BLOCK_HIGH(worldRank, n-1, worldSize);
```

- Calcular o valor inicial a ser marcado, sendo este o valor mínimo do bloco se múltiplo do número primo atual, senão será o múltiplo mais próximo.
- No final de cada iteração do ciclo, enviar a todos os processos o valor do próximo número primo.

```
1     for (long k = 2;k*k <= n;) {
2         if(k*k < blockLow){
3             if(blockLow % k == 0){
4                 startBlockValue = blockLow;
5             }else{
6                 startBlockValue = (blockLow + (k-(blockLow % k)));
7             }
8         }else{
9             startBlockValue = k*k;
10        }
11
12        // Mark as false all multiples of k between k*k and n
13        for (unsigned long long i = startBlockValue; i <= blockHigh; i += k){
14            primes[i-blockLow] = true;
15        }
16
17        // Set k as the smaller urmarked number > k
18        if(worldRank == ROOT){
19            do {
20                k++;
21            } while (primes[k - blockLow] && k*k < blockHigh);
22        }
23
24        MPI_Bcast(&k, 1, MPI_LONG, ROOT, MPI_COMM_WORLD);
25    }
```

4.3 Versão híbrida com memória partilhada e distribuída

Finalmente, esta última implementação é um paralelismo híbrido, pois aproveita ambas as melhorias existentes nas duas versões acima, usando tanto a API **OpenMP** como a biblioteca **OpenMPI**. Tendo como base a versão com memória distribuída, adiciona-se o seguinte excerto de código proveniente da versão com memória partilhada:

```
1    // Mark as false all multiples of k between k*k and n
2    #pragma omp parallel for num_threads(n_threads)
3    for (unsigned long long i = startBlockValue; i <= blockHigh; i += k){
4        primes[i-blockLow] = true;
5    }
```

4.4 Tempo de execução das implementações paralelas

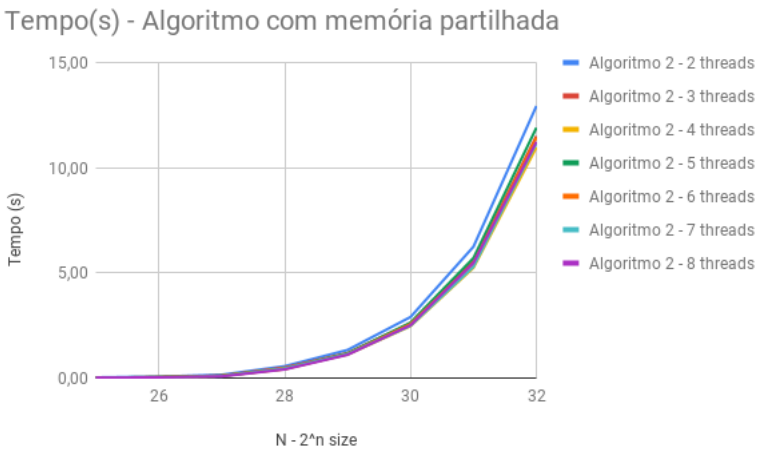

Figura 4: Tempo de execução do algoritmo 2

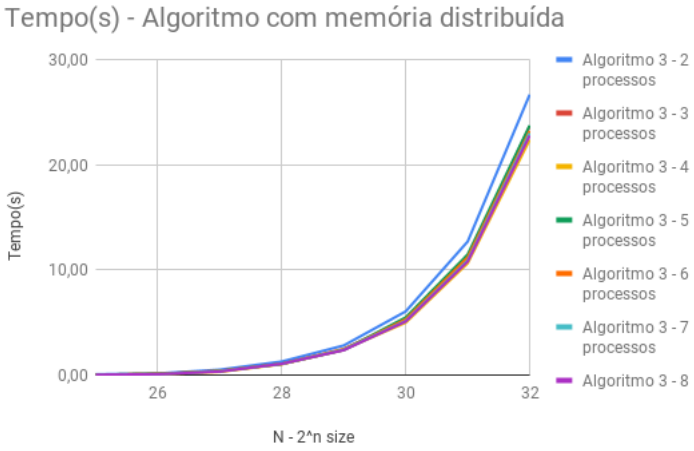

Figura 5: Tempo de execução do algoritmo 3

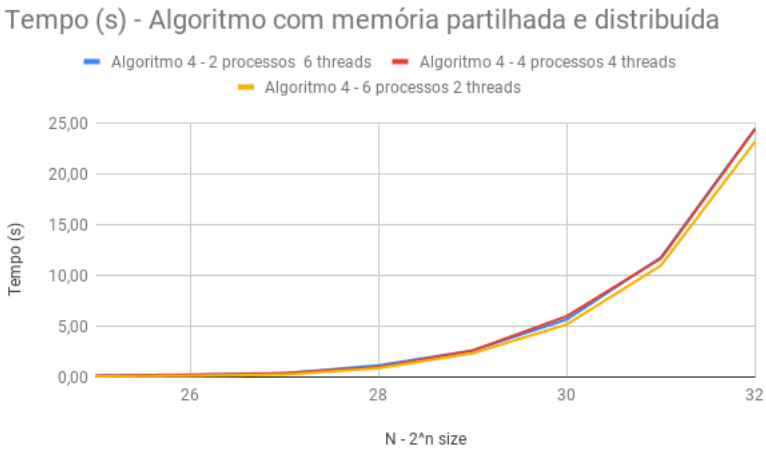

Figura 6: Tempo de execução do algoritmo 4

Para todos os algoritmos paralelizados pode ser feita a mesma conclusão que foi feita na versão sequencial. O tempo decorrido aumenta consoante o aumento do n e que esse aumento é próximo do fator 2, o que corresponde também ao aumento da dimensão do intervalo de primos a serem gerados.

5 Resultados e análise

5.1 Métricas de desempenho e metodologia da avaliação

Todos os algoritmos foram implementados e testados em C++. Para a obtenção de resultados utilizaram-se intervalos de grande escala, nomeadamente, potências de 2 com expoente a variar de 25 a 32. Para todas as experiências foram feitas 3 medições de tempo de execução sendo considerada a média destes valores para a análise e comparação. Nas implementações paralelizadas foram adicionados outros fatores como o número de processos a utilizar, no caso do algoritmo com memória distribuída (OpenMPI), o número de *t*hreads, no caso do algoritmo com memória partilhada (OpenMP) e ambos os anteriores no caso do algoritmo híbrido. Cada experiência foi realizada em apenas um computador sendo permitida a variação do número de processos e *t*hreads entre 2 e 8 inclusive. As métricas de desempenho utilizadas foram as seguintes:

- Tempo de execução - Tempo necessário à execução de uma experiência.
- *M*OP/s - Milhões de instruções por segundo em função do número máximo da dimensão dos dados (n), permite avaliar a velocidade média de processamento;

$$\text{MOP/s} = \frac{n*log(log(n))}{Ti*10^6}$$

- *S*peedUp - Permite avaliar o quanto a versão paralela será melhor/pior a nível do tempo de execução relativamente à versão sequencial do algoritmo em função do número máximo da dimensão dos dados (n).

$$\text{SpeedUp} = \frac{Tsequencial(n)}{Tparalelo(n)}$$

- Eficiência - Em função do número de *p*rocessadores disponíveis, a eficiência é o rácio de utilização dos processadores (P) na execução do programa em paralelo, esta permite avaliar a escalabilidade dos algoritmos paralelizados.

$$\text{Eficiência} = \frac{SpeedUp}{p}$$

Através destas métricas de desempenho será será possível comparar e avaliar a qualidade as várias versões paralelas implementadas, quantificar a sua superioridade/inferioridade em relação à versão sequencial e ainda a sua escalabilidade.

Para obtenção dos resultados foi utilizado um computador com um processador (Intel) i7-4790, 16GB de memória RAM e uma otimização -O3 ao compilar os programas.

5.2 Análise dos Resultados

5.2.1 Tempo de execução e MOP/s

Através da análise das figuras 3, 4, 5 e 6 apresentadas em capítulos anteriores verifica-se que, contrariamente ao expectável, o algoritmo 3 tem um tempo de execução maior que o algoritmo 1, ou seja, a versão paralelizada com OpenMPI é mais lenta que a versão sequencial. Como consequência a implementação híbrida é a que tem tempos de execução mais reduzidos comparativamente às outras versões, sendo esta implementação baseada na versão com memória distribuída. Isto deve-se a certas limitações relativamente à(s) máquina(s) onde é executada, à comunicação entre processos, e ao excesso de processos e/ou *threads* que podem levar a que o tempo de execução aumente exponencialmente. No entanto, o tempo de execução do algoritmo 2 encontra-se dentro do expectável sendo este menor do que o da versão sequencial.

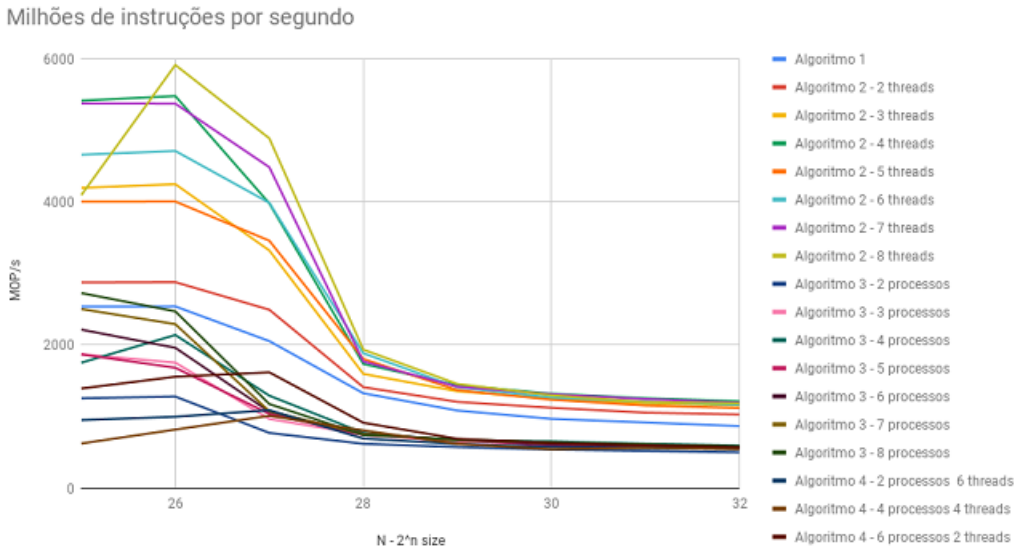

Figura 7: MOP/s de todos os algoritmos

No que diz respeito à performance dos algoritmos, podemos verificar para todos eles um decréscimo no número de operações por segundo - Figura 7. Esse decréscimo deve-se a uma gestão de memória mais intensiva, consequente do aumento exponencial do tamanho da lista de primos a marcar. Também podemos verificar aqui o que já concluímos na comparação dos tempos de execução, sendo os algoritmos com memória distribuída (algoritmo 3 e 4) os com menor valor de MOP/s em relação aos restantes, isto claro, corresponde ao facto de estes terem um maior tempo de execução. Da mesma forma podemos verificar

que algoritmo com memória partilhada (algoritmo 2) obteve um maior valor de MOP/s em relação ao sequencial.

5.2.2 Análise de desempenho

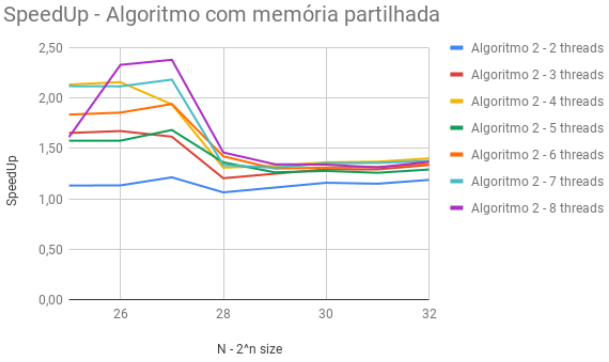

Figura 8: *SpeedUp* do algoritmo 2

Com base na Figura 8, conclui-se que o *SpeedUp* calculado, em comparação com a versão sequencial, apresenta geralmente melhores resultado com o uso de 4 *threads* e que a partir deste valor o aumento do número de *threads* não se traduz num benefício temporal. No entanto, para valores de $n < 28$, quando a gestão de memória não é tão intensa, podemos verificar que a utilização de um número *threads* 7 e 8 pode apresentar melhores resultados. Note-se ainda que, quando o número de *threads* utilizadas é superior a 2, o *SpeedUp* tende para um valor $\in [120\%, 130\%]$.

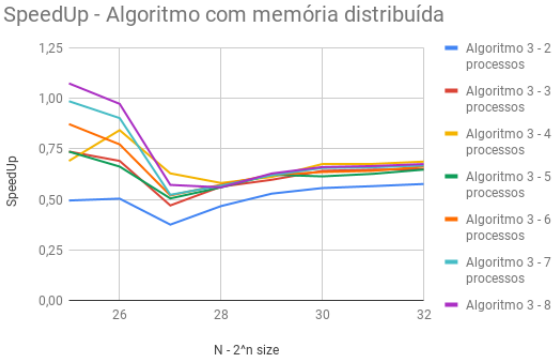

Figura 9: *SpeedUp* do algoritmo 3

Para o algoritmo 3, após analisar os valores de *SpeedUp* obtidos e os tempos de execução do algoritmo, é possível verificar que, como esperado, a paralelização do processo com o algoritmo de memória distribuída apresenta valores de *SpeedUp* bastante inferiores aos obtidos com o algoritmo de memória partilhada, sendo este inferior a 1 para a maioria das experiências deste algoritmo. Isto significa que, seja qual for número de processos no mesmo computador, este algoritmo é mais lento do que a versão sequencial do mesmo.

Por outro lado, tal como verificámos no algoritmo 2, este algoritmo apresenta habitualmente melhores resultados com a utilização de 4 processos. Podemos também concluir que o aumento do número de processos não leva ao aumento do *SpeedUp* para valores de n $>$ 27, ou seja, quando a gestão de memória começa a ser mais intensa.

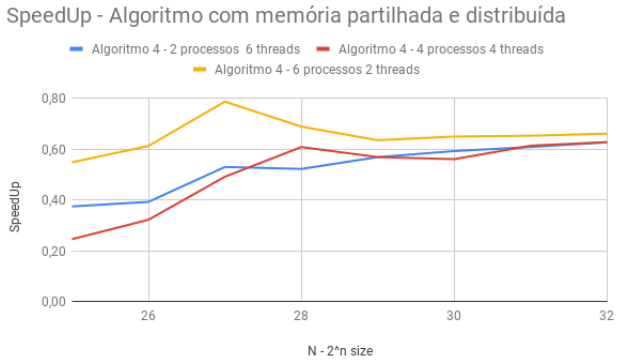

Figura 10: *SpeedUp* do algoritmo 4

Para o algoritmo 4, podemos verificar que, conjugando as versões do algoritmo de memória partilhada e memória distribuída, não obtivemos valores de *SpeedUp* superiores aos da versão do algoritmo com memória partilhada. Também podemos verificar que o *SpeedUp*, para este algoritmo, continua inferior a 1, dado que este algoritmo herda todos os problema do algoritmo 3. Para além disso, a utilização de um número de processos + número de *threads* superior à capacidade do processador não se traduz em nenhum benefício.

5.2.3 Análise da Escalabilidade

A análise da escalabilidade permite, através da eficiência, verificar a qualidade da paralelização, isto é, permite perceber se o algoritmo mantém a mesma qualidade (mesmo rácio de qualidade) aumentando o número de processadores e o volume de dados. Os gráficos seguintes apresentam os valores do estudo efetuado para a versão paralela dos algoritmos de memória distribuída e híbrida.

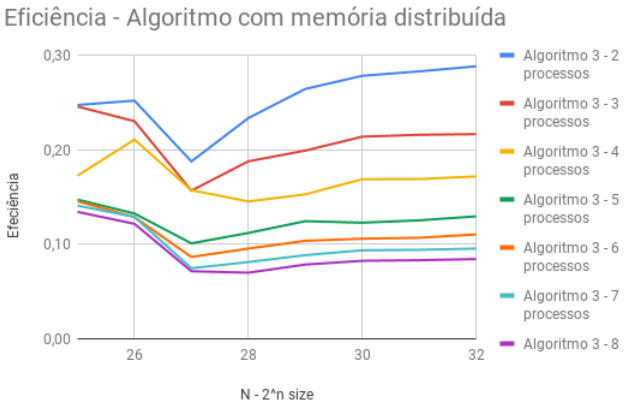

Figura 11: Eficiência do algoritmo 3

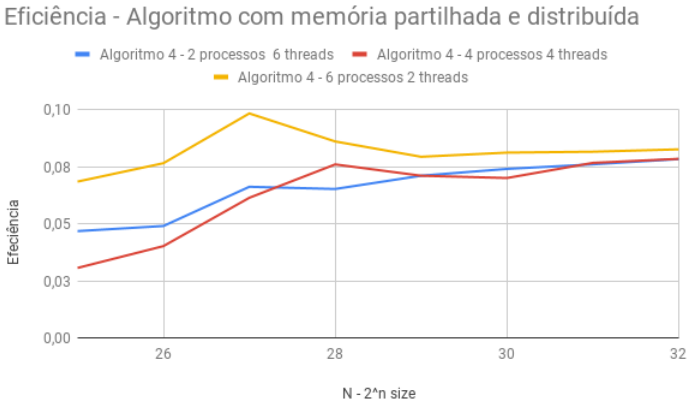

Figura 12: Eficiência do algoritmo 4

Através da interpretação dos dois gráficos anteriores, os valores de eficiência diminuem com o aumento do número de processadores utilizados, mas mantém-se relativamente constantes com o aumento do volume de dados.

Apesar de todos os problemas mencionados anteriormente com os algoritmos que fazem utilização de memória distribuída, podemos concluir que estes são escaláveis a problemas maiores, uma vez que os valores apresentados estão, maioritariamente, no intervalo [0, 1]. No entanto, tendo em conta que apenas foi utilizado um computador para obter estes resultados, estes valores são inconclusivos.

6 Conclusão

Com a realização deste projeto, foi possível aplicar os conhecimentos adquiridos relativamente ao paralelismo entre processos e/ou *threads*, bem como aprofundar o contacto inicial com a API **OpenMP** e a biblioteca **OpenMPI**

Tendo em conta os resultados que foram obtidos, a versão com memória partilhada é a que oferece um melhor tempo de execução. Isto pode ser explicado pelo facto de, tanto nos sistemas com memória distribuída como nos de memória híbrida, o custo computacional do algoritmo não ser alto o suficiente para compensar o custo do *broadcast* da variável k entre processos.

Para trabalho futuro, uma possível melhoria seria a implementação de uma versão de memória distribuída onde cada processo calcula os números primos até à raiz quadrada de N, pois este cálculo não tem um grande custo computacional e consequentemente não seria necessária a comunicação entre os diferentes processos. Para além disto, devem também ser feitas outras experiências com mais do que um computador, ou seja, testar a biblioteca **OpenMPI** com um maior número de processos e processadores.