Avaliação de Performance

Relatório Trabalho 2

Computação Paralela

Mestrado Integrado em Engenharia Informática e Computação

Elaborado por:
Pedro Filipe Agrela Faria - 201406992

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

4 de Março de 2018

Conteúdo

1- Descrição do problema

Em resposta ao repto lançado pelo docente da unidade curricular de Computação Paralela, desenvolvi pequenas aplicações em C++ tendo como objetivo o estudo da performance e escalabilidade em dois modelos de memória de programação paralela.

Para resolver e analisar o problema, foi sugerido pelo docente o uso de dois algoritmos (Decomposição em LU e Crivo de Eratóstenes), fazendo uso de bibliotecas como o OpenMP, que faz manuseamento de memória partilhada, e o MPI, que faz manuseamento de memória distribuída.

O objetivo do primeiro algoritmo, Decomposição em LU, é repartir uma matriz quadrática no produto de duas matrizes, sendo estas, uma matriz triangular inferior e uma matriz triangular superior.

O objetivo do segundo algoritmo, Crivo de Eratóstenes, é no cálculo de números primos num dado intervalo entre 2 a N, sendo N um número natural igual ou superior a 2. Um número é considerado primo se for apenas divisível por sí próprio e por 1.

2- Soluções Sequenciais e medidas de desempenho

2.1- Algoritmo 1 - Decomposição LU

O objetivo deste algoritmo é factorizar uma matriz quadrática no produto de duas matrizes LU. A factorização usada neste algoritmo foi a de Doolittle. A complexidade temporal deste é de $O(N^3)$ uma vez que é usado 3 loops (1 loop para cada for). A seguinte figura representa como é feito o preenchimento da matriz superior e inferior:

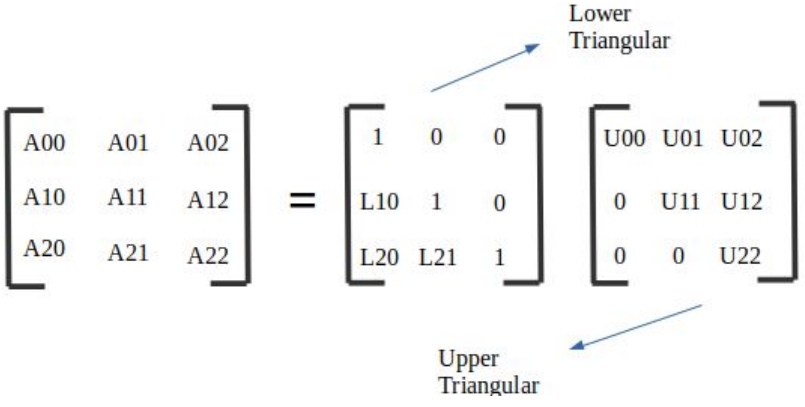

Figura 1: Cálculo do algoritmo 1.

O seguinte pseudocódigo representa o algoritmo aplicado:

```
for i = 1, ···, n
    for j = 1, ···, i − 1
        α = a_ij
        for p = 1, ···, j − 1
            α = α − a_ip a_pj
        a_ij = α / a_jj
    for j = i, ···, n
        α = a_ij
        for p = 1, ···, i − 1
            α = α − a_ip a_pj
        a_ij = α
```

Figura 2: Pseudocódigo do algoritmo 1.

2.2- Algoritmo 2 - Crivo de Eratóstenes

Este algoritmo trabalha sobre um vetor do tamanho N-1, sendo N o número de números primos a verificar. O vetor é inicializado todo como verdadeiro e ao fazer o cálculo até $\sqrt{N}$ irá atualizar as posições deste vetor como falso os números que não são primos. Este algoritmo possui complexidade temporal $O(n \log \log n)$. A seguinte imagem ilustra a marcação dos números primos de 2 a 100:

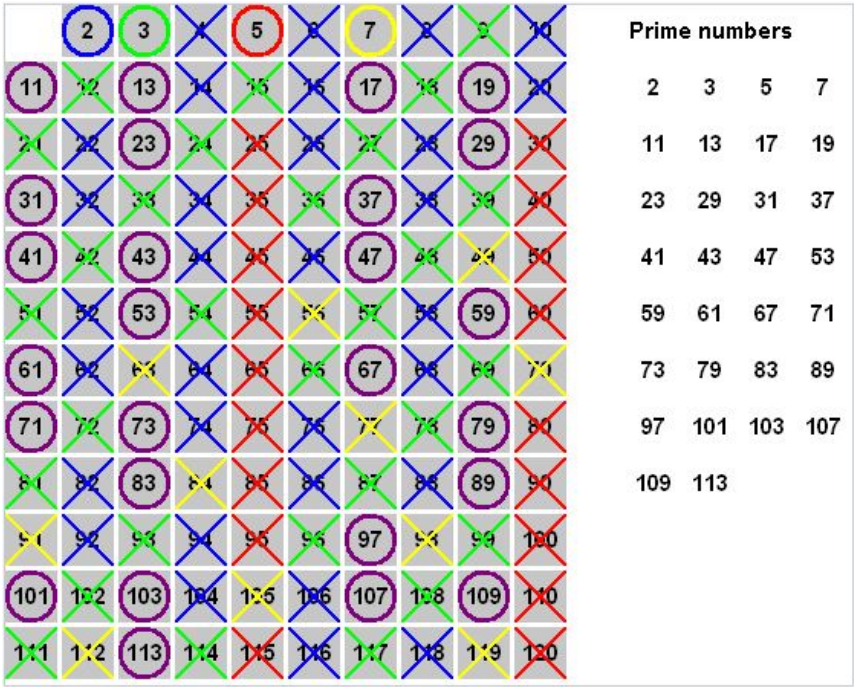

Figura 3: Ilustração do algoritmo 2.

O seguinte pseudocódigo representa o algoritmo aplicado:

```
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 4: Pseudocódigo do algoritmo 2.

2.3- Medidas de desempenho

As medidas utilizadas foram o número de *threads*, tempo de execução, tamanho da matriz (no caso do algoritmo 1), número de primos a verificar (no caso do algoritmo 2). Foi ainda calculado métricas derivadamente, sendo este, as operações por vírgula flutuante (FLOPS), no qual a fórmula usada foi: $\frac{TamanhoMatriz^3}{TempoDeExecução}$ para o algoritmo 1, já no algoritmo 2 foi usada a seguinte fórmula: $\frac{Nr_{Primos} \log \log Nr_{Primos}}{TempoDeExecução}$.

A metodologia de avaliação aplicada neste trabalho foi na comparação dos resultados obtidos. Através destas comparações de resultados entre os tempos de execução, a mudança do número de *threads*/processos, ao cálculo dos FLOPS, foi possível chegar à conclusão de que cada uma destas métricas ajudam na obtenção de resultados finais distintos.

Para obtenção destes resultados foi utilizado um processador Intel Core i5 dual-core a 2,0 GHz, Turbo Boost até 3,1 GHz, com 32KB de cache L1, 256KB de cache L2 e 4MB de cache L3 e ainda, uma otimização O3 ao compilar o programa em C++ de forma a obter os melhores resultados possíveis.

3- Soluções paralelizadas

Sendo o objetivo do trabalho, em estudar a performance e a escalabilidade dos algoritmos, foram implementadas duas versões diferentes de modelo de memória de programação paralela. No qual o OpenMP, faz um manuseamento de memória partilhada, e o MPI, faz um manuseamento de memória distribuída. Na versão de memória partilhada, vários processos executam de forma paralela partilhando os mesmos recursos de memória, já na versão de memória distribuída a memória não é partilhada no qual os processos possuem a sua própria memória

local, no qual o programador tem a decisão de como a memória é repartida pelos diferentes processos e como estes são sincronizados.

3.1- Algoritmo 1 - Decomposição LU

3.1.1- Modelo de Memória Partilhada

No modelo de memória partilhada do algoritmo 1 foi usada a biblioteca OpenMP para a inclusão do pragma *opm parallel for* de forma a dividir sequencialmente as iterações dos ciclos for entre um número de threads (n_threads) recebidas. O seguinte código mostra como foi usado esse pragma:

```
#pragma omp parallel for private(i, k) num_threads(n_threads)
for(j=0; j<n; j++){
    for(i=0; i<n; i++){
        if(i<=j){
            u[i][j]=a[i][j];
            for(k=0; k<=i-1; k++)
                u[i][j]-=l[i][k]*u[k][j];
            if(i==j)
                l[i][j]=1;
            else
                l[i][j]=0;
        }else{
            l[i][j]=a[i][j];
            for(k=0; k<=j-1; k++)
                l[i][j]-=l[i][k]*u[k][j];
            l[i][j]/=u[j][j];
            u[i][j]=0;
        }
    }
}
```

Figura 5: Código do algoritmo 1 usando memória partilhada.

3.1.2- Modelo de Memória Distribuída

No modelo de memória distribuída do algoritmo 1 foi usada a biblioteca MPI. O algoritmo trata de fazer os cálculos na matriz de entrada, sendo esta, atualizada a cada passo do cálculo. Este começa por apontar para o elemento da linha pivô que pertença à diagonal, iterando assim sobre todos os coeficientes da coluna pivô da matriz. O algoritmo garante também a divisão entre os seus processos e é usado o MPI_Bcast para garantir a sincronização entre os processos.

A seguinte figura, ilustra a secção de código usado para o cálculo da matriz LU no modelo de memória distribuída:

```
for(i = 0; i < n-1; i++, ref_diag++){
    float* row_diag = &a[ref_diag * n + ref_diag];

    for(j = ref_diag + 1; j < n; j++){

        if(j % world_size == world_rank){
            float* back = &a[j * n + ref_diag];

            if(*back == 0)
                return;

            float k = *back / row_diag[0];
            int w;

            for (w = 1; w < n - ref_diag; w++) {
                (back)[w] = (back)[w] - k * row_diag[w];
            }
            *back = k;
        }
    }

    for(j = ref_diag+1; j < n; j++){
        float* back = &a[j * n + ref_diag];
        MPI_Bcast(back, n-ref_diag, MPI_FLOAT, j % world_size, MPI_COMM_WORLD);
    }
}
```

Figura 6: Código do algoritmo 1 usando memória distribuída.

3.2- Algoritmo 2 - Crivo de Eratóstenes

3.2.1- Modelo de Memória Partilhada

No modelo de memória partilhada do algoritmo 2 foi usada a biblioteca OpenMP para a inclusão do pragma *opm parallel for* de forma a dividir sequencialmente as iterações do ciclo for entre um número de threads (n_threads) recebidas. O seguinte código mostra como foi usado esse pragma:

```
while (k*k <= n) {

    // Mark as false all multiples of k between k*k and n
    #pragma omp parallel for num_threads(n_threads)
    for (i = k*k; i <= n; i += k)
        values[i-2] = false;

    // Set k as the smaller urmarked number > k
    for(i = k+1; i <= n; i++){
        if (values[i-2]) {
            small = i;
            break;
        }
    }
    k = small;
}
```

Figura 7: Código do algoritmo 2 usando memória partilhada.

3.2.2- Modelo de Memória Distribuída

No modelo de memória distribuída do algoritmo 2 foi usada a biblioteca MPI. Tal como na versão sequencial, o vector é inicializado com valores verdadeiros em cada posição no qual é dividido por p processos, no qual cada processo i, fica atribuído a um bloco de dados cujo o primeiro elemento é calculado através da seguinte fórmula: $BK_{LOW}(i,n,p) = \frac{i*n}{p}$, já o último elemento é calculado pela fórmula: $BK_{HIGH}(i,n,p) = BK_{LOW}(i+1,n,p) - 1$, e o tamanho do bloco no vetor em cada p processos é calculado pela seguinte fórmula: $BK_{SIZE}(i,n,p) = BK_{LOW}(i+1,n,p) - BK_{LOW}(i,n,p)$. A seguinte imagem ilustra as suas definições no código C++:

```
#define BK_LOW(i,n,p) ((i)*(n)/(p))
#define BK_HIGH(i,n,p) (BK_LOW((i)+1,n,p)-1)
#define BK_SIZE(i,n,p) (BK_LOW((i)+1,n,p)-BK_LOW(i,n,p))
```

Figura 8: Definições usadas no algoritmo 2 para o modelo de memória distribuída.

Uma vez que só é necessário marcar os múltiplos de k, cada processo i precisa de verificar se esses múltiplos precisam de ser marcados nesse seu bloco atribuído. A primeira condição (if) dentro do ciclo (while) da seguinte figura, trata de fazer essa verificação. O processo principal (world_rank=0), trata de marcar os múltiplos de k no bloco de dados e verifica qual será o próximo valor de k, fazendo broadcast, através da função MPI_Bcast, para os processos seguintes. A seguinte figura, ilustra a secção de código usado para o cálculo dos números primos no modelo de memória distribuída:

```
while (k*k <= n) {

    if(k*k < low_value){
        if(low_value % k == 0){
            j = low_value;
        }else{
            j = (low_value + (k-(low_value % k)));
        }
    }else{
        j = k*k;
    }

    // Mark as false all multiples of k between k*k and n
    for (i = j; i <= high_value; i += k){
        values[i-low_value] = false;
    }

    // Set k as the smaller urmarked number > k
    if(world_rank == root){
        for(i = k+1; i < high_value; i++){
            if (values[i-low_value]) {
                k = i;
                break;
            }
        }
    }
    MPI_Bcast(&k, 1, MPI_INT, root, MPI_COMM_WORLD);
}
```

Figura 9: Código do algoritmo 2 usando memória distribuída.

4- Resultados obtidos

4.1 Algoritmo 1

4.1.1 - Versão Sequencial

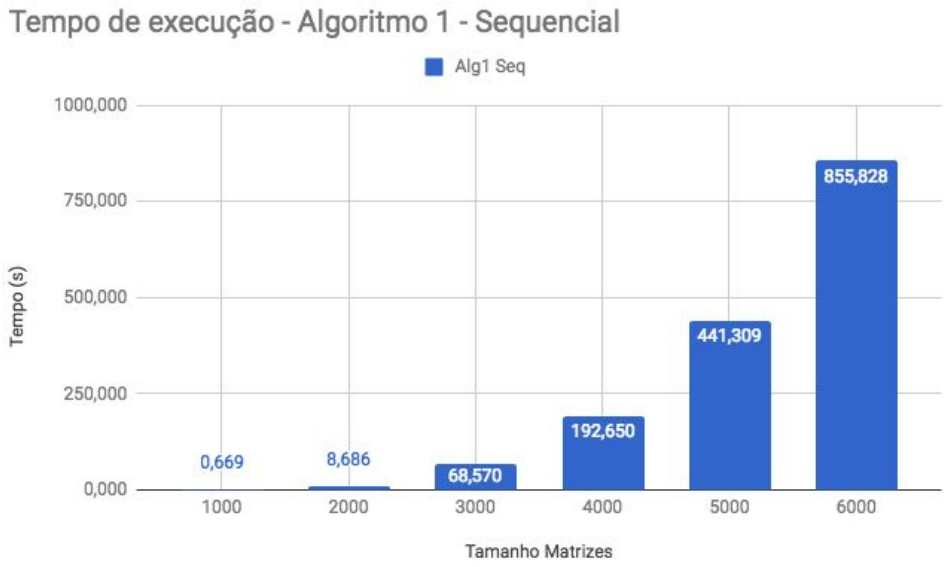

Figura 10: Gráfico temporal do algoritmo 1 - Sequencial.

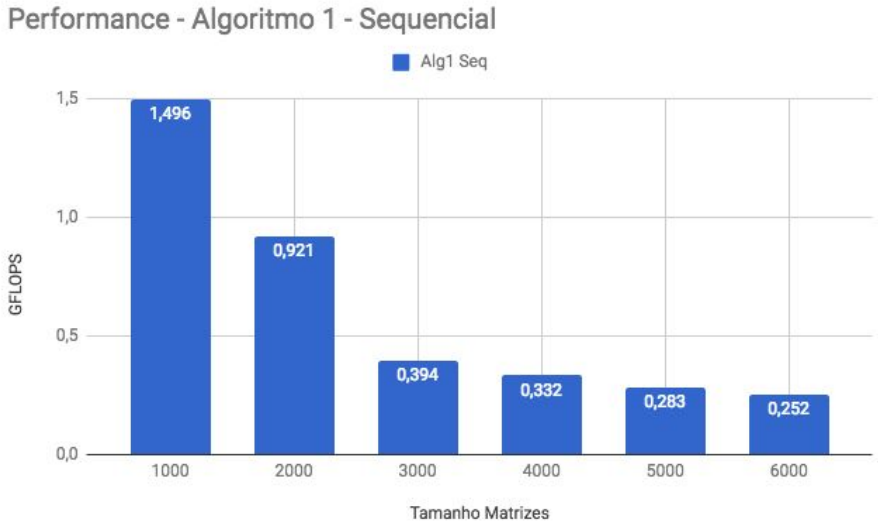

Figura 11: Gráfico performance do algoritmo 1 - Sequencial.

As figuras 10 e 11 representam, respetivamente, o tempo de execução e o número de FLOPS, do algoritmo 1 programados e executados em C++. Foram testados valores para os diferentes tamanhos das matrizes, entre 1000 a 6000. Pela

figura da performance (Figura 11), é possível verificar que o número de FLOPS começa a estabilizar quando o tamanho da matriz é de 3000 a 6000, com um valor de 0,3 GFLOPS.

4.1.2 - Modelo de Memória Partilhada/Distribuída

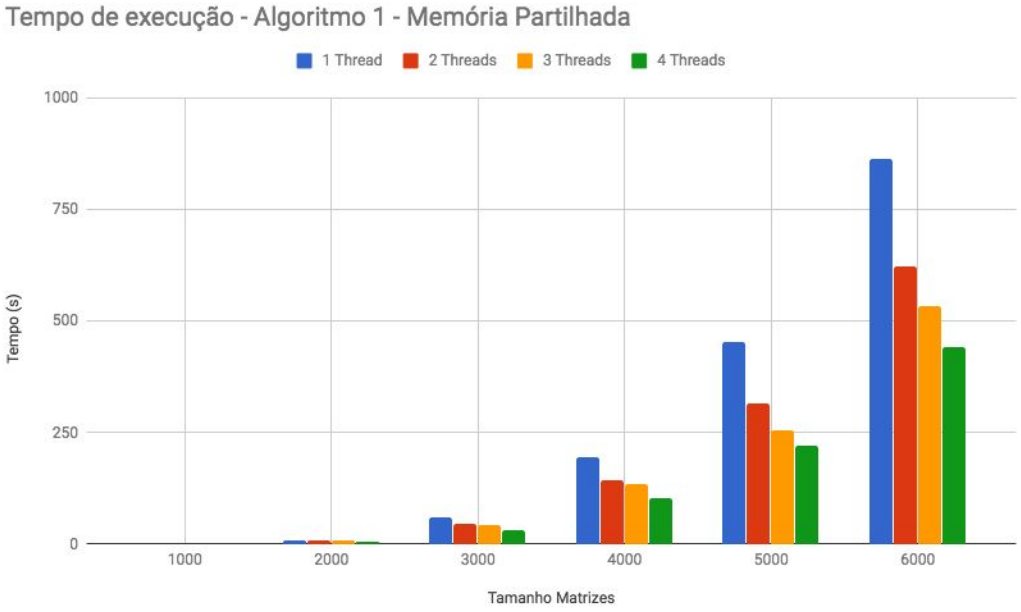

Figura 12: Gráfico temporal do algoritmo 1 - Memória Partilhada.

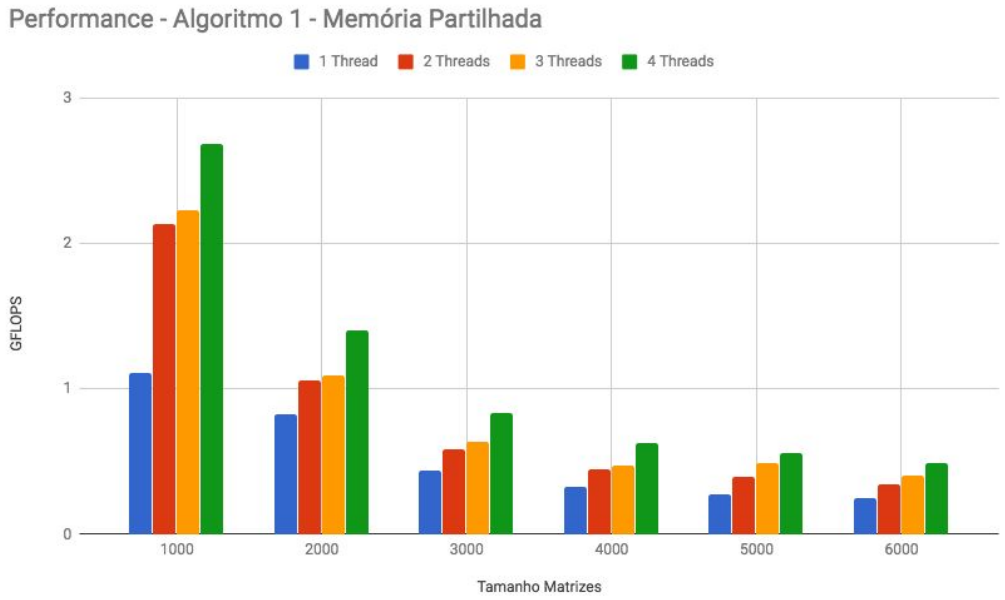

Figura 13: Gráfico performance do algoritmo 1 - Memória Partilhada.

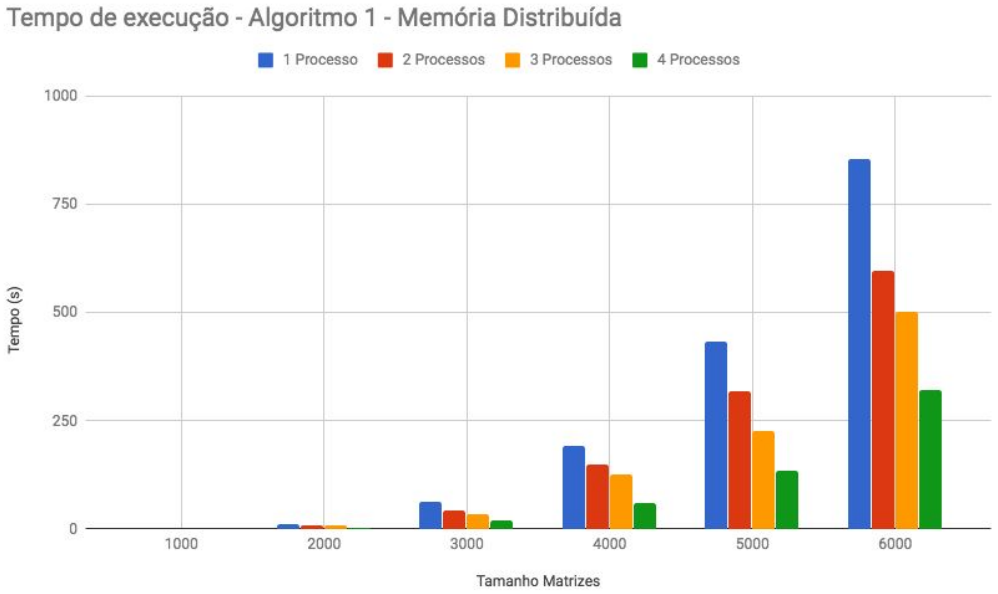

Figura 14: Gráfico temporal do algoritmo 1 - Memória Distribuída.

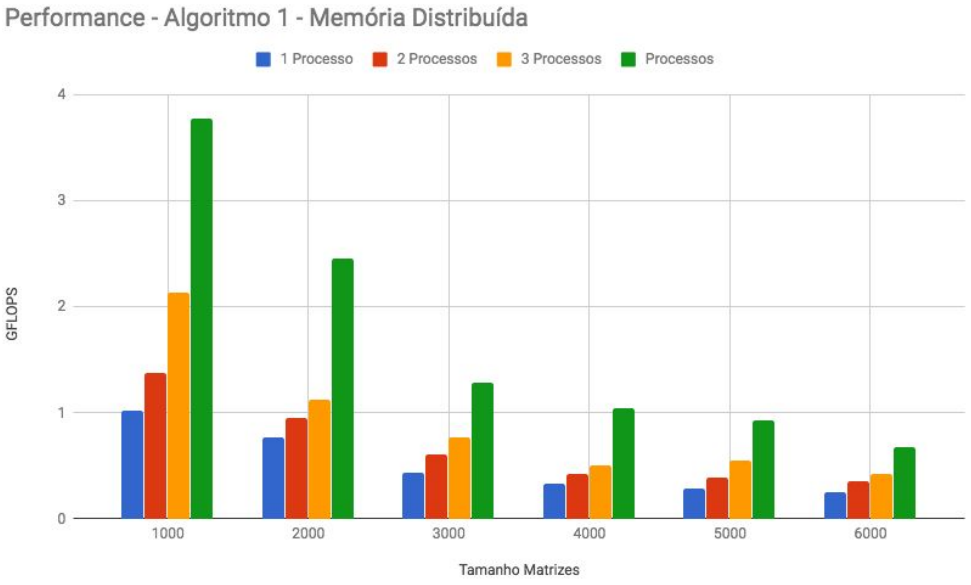

Figura 15: Gráfico performance do algoritmo 1 - Memória Distribuída.

Em relação ao modelo de acesso de memória partilhada podemos verificar nas figuras 12 e 13, que há uma estabilização da performance quando o tamanho da matriz tem um tamanho de 3000, com uma performance média de 0,49 GFLOPS. De realçar também que no uso de 2 ou mais threads houve uma melhoria considerável nos tempos de execução.

Quanto ao modelo de acesso de memória distribuída, podemos verificar nas figuras 14 e 15, que no uso de 1 a 3 processos a performance não variou muito e

este começa também a estabilizar quando a matriz tem um tamanho de 3000. É importante referir que no uso de 4 processos em relação ao uso de 3 processos, existe uma melhoria média de 40%.

Comparando os dois tipos de modelos de acesso, podemos verificar que a memória distribuída tem uma melhor performance que a memória partilhada no uso de 3 ou 4 processos. Já a memória partilhada, consegue ter uma melhor performance no uso de 1 ou 2 processos.

4.2 Algoritmo 2

4.2.1 - Versão Sequencial

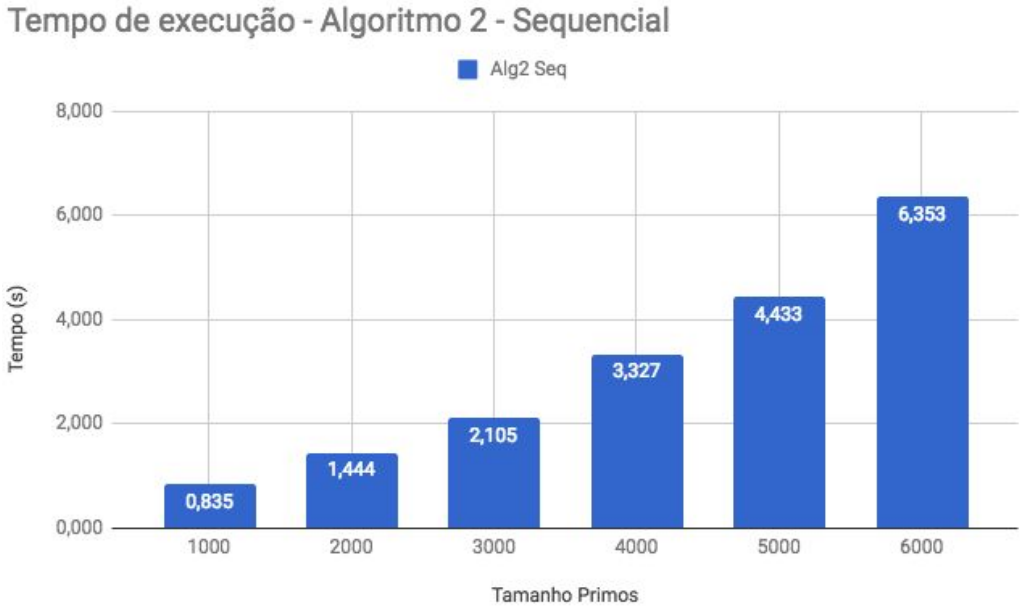

Figura 16: Gráfico temporal do algoritmo 2 - Sequencial.

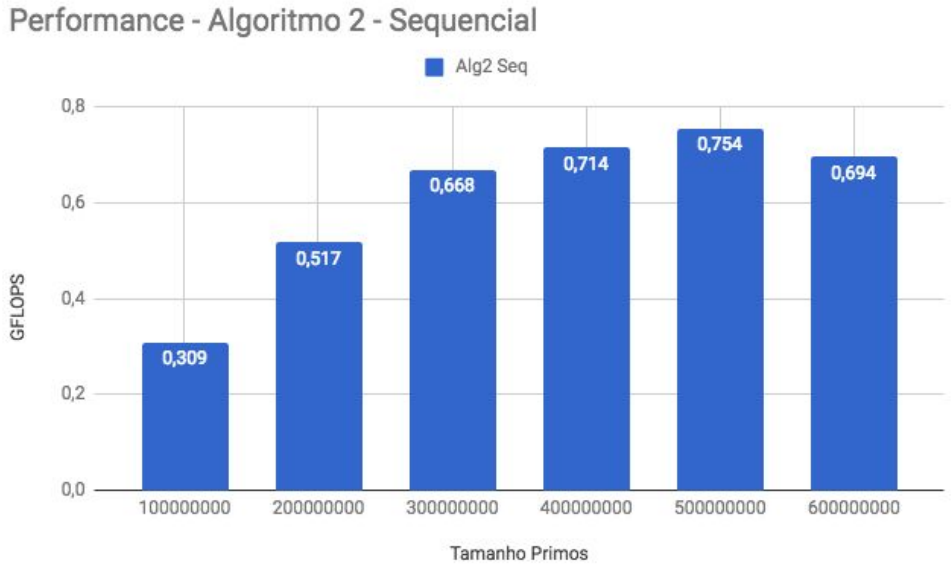

Figura 17: Gráfico performance do algoritmo 2 - Sequencial.

As figuras 16 e 17 representam, respetivamente, o tempo de execução e o número de FLOPS, do algoritmo 3 programados e executados em C++. Foram testados valores para os diferentes tamanhos de números primos a serem verificados, entre 100000000 a 600000000. Pela figura da performance (Figura 17), é possível verificar que o número de FLOPS começa a estabilizar quando o número de primos a verificar é de 300000000 a 600000000, com um valor de aproximadamente 0,7 GFLOPS.

4.2.2 - Modelo de Memória Partilhada/Distribuída

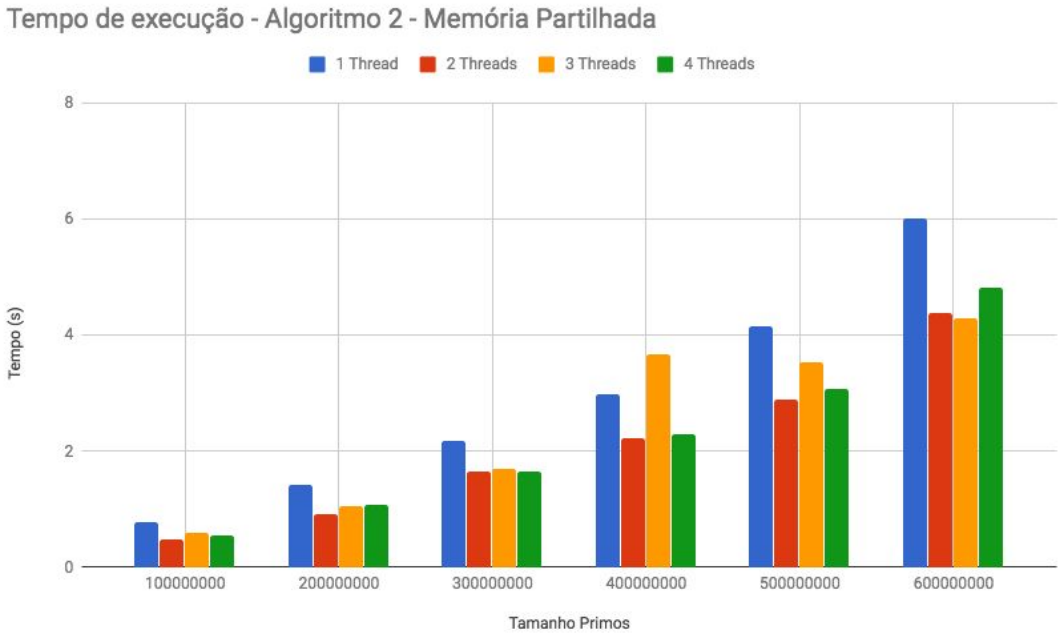

Figura 18: Gráfico temporal do algoritmo 2 - Memória Partilhada.

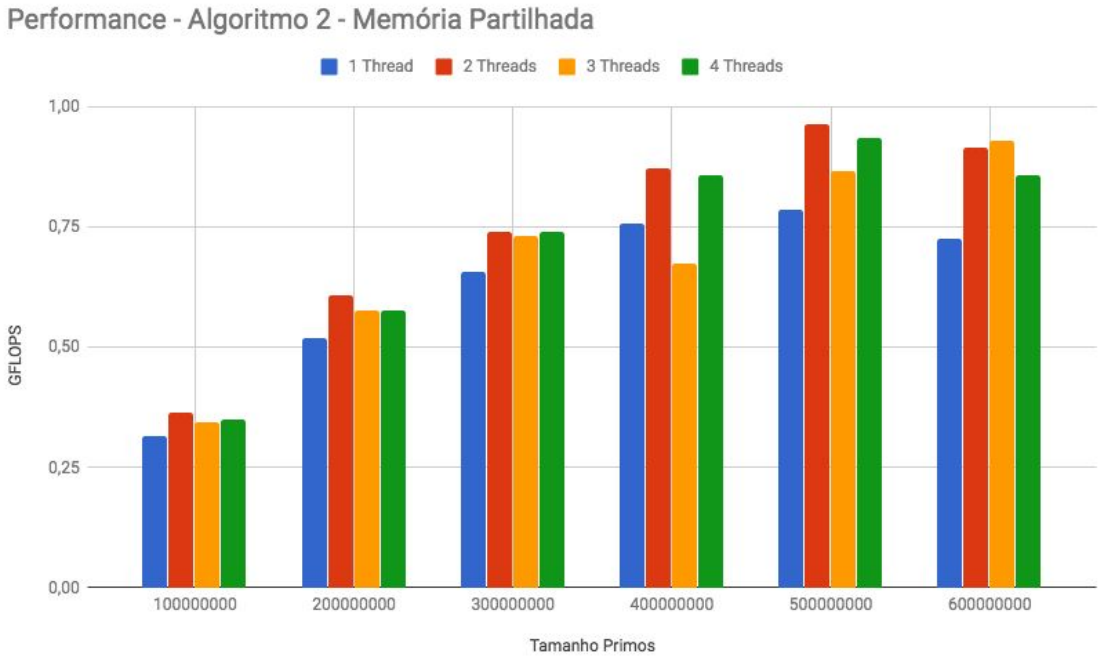

Figura 19: Gráfico performance do algoritmo 2 - Memória Partilhada.

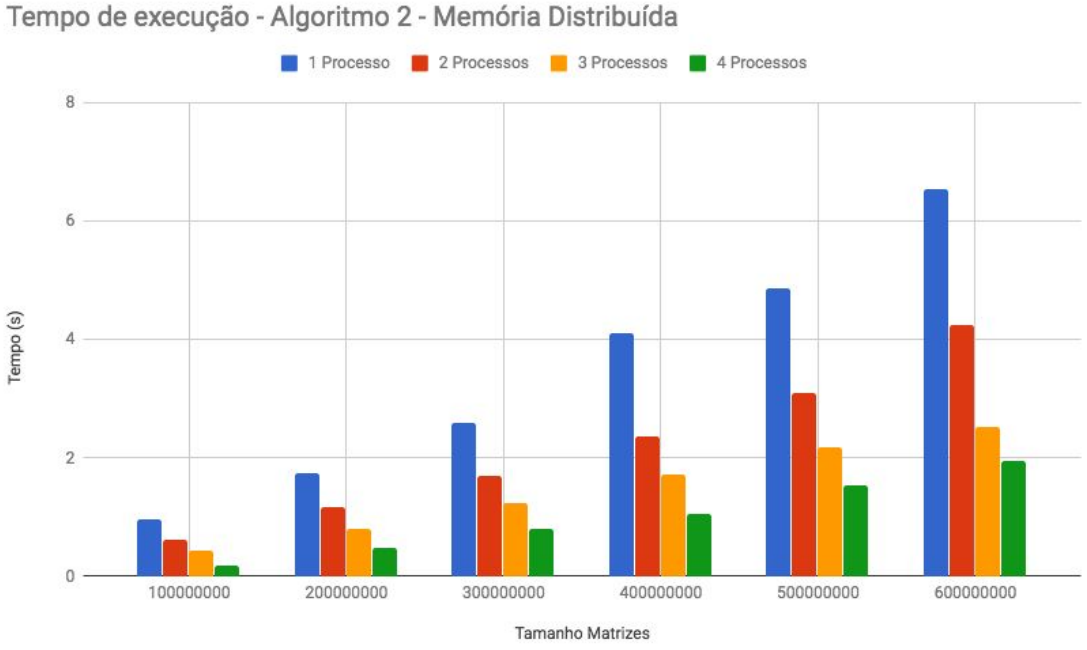

Figura 20: Gráfico temporal do algoritmo 2 - Memória Distribuída.

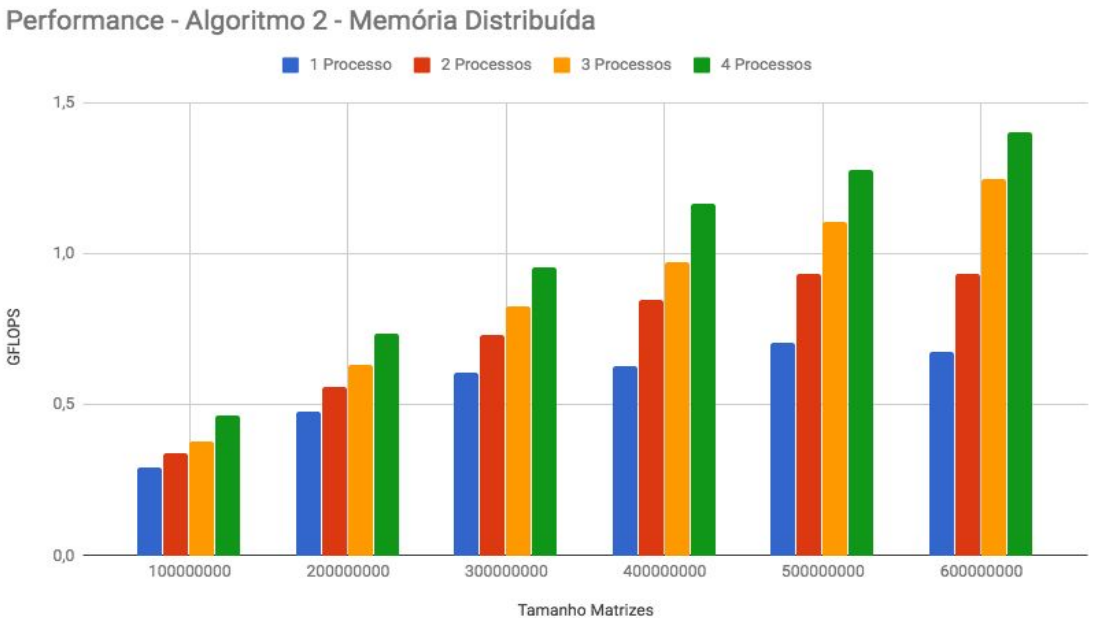

Figura 21: Gráfico performance do algoritmo 2 - Memória Distribuída.

Em relação ao modelo de acesso de memória partilhada podemos verificar nas figuras 18 e 19, que o uso de 3 e 4 threads tende a ter uma menor performance que o uso de 2 threads e este começa a estabilizar quando o número de primos a serem verificados é de 500000000.

Quanto ao modelo de acesso de memória distribuída, podemos verificar nas figuras 20 e 21, que há uma tendência de crescimento quase linear, com o aumento do tamanho do número de primos a serem verificados, isto com 2 a 4 processos.

Comparando os dois tipos de modelos de acesso, podemos verificar que ao contrário da memória partilhada, que começa a baixar a performance no uso de 3 a 4 processos, a memória distribuída consegue melhorar a performance no uso dessa mesma quantidade de processos, conseguindo um aproveitamento de cerca de 60%.

5- Conclusões

Este projeto teve como objetivo a análise da performance e escalabilidade em dois modelos de memória de programação paralela.

É possível afirmar que o uso do modelo de memória distribuída obteve melhores resultados que o modelo de memória partilhada, especialmente quando o uso de 3 ou mais processos.

De realçar que houve uma dificuldade acrescida no uso da biblioteca MPI, uma vez que cabe ao utilizador fazer gestão dos dados, dividindo-os por processos e sincronização dos mesmos.