U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

# Relatório Projeto 2

Computação Paralela

2018/2019

Mestrado Integrado em
Engenharia Informática e Computação

João Francisco Veríssimo Dias Esteves
José Pedro Dias de Almeida Machado

18 de Maio de 2019

# Conteúdo

# 1 Introdução

Este trabalho tem o propósito de avaliar a melhoria obtida em termos de tempo de execução com a paralelização do algoritmo *Sieve of Eratosthenes* que é utilizado para encontrar números primos até um dado número dado como parâmetro. O problema de cálculo de números primos muito grandes é um problema atual porque os mesmo são utilizados na encriptação, por exemplo no algoritmo RSA a chave pública é calculada pelo produto de dois números primos muito grandes e a chave secreta consiste nos próprios primos. Quanto maiores forem os números primos na chave pública maior a segurança da mesma mas também maior poder computacional necessário para os calcular.

# 2 Descrição do Problema

Neste segundo trabalho o problema a ser analisado é a paralelização do algoritmo *Sieve of Eratosthenes*, fazendo duas versões, uma utilizando *OpenMP* e outra utilizando *MPI*. Como ponto de referência, é usada também uma versão sequencial, simples, do algoritmo.

# 3 Sieve of Eratosthenes

Este algoritmo é um algoritmo simples para encontrar números primos até um dado número começando por marcar por marcar os múltiplos de 2 como não sendo primos, depois busca o número mais baixo ainda não marcado e marca os múltiplos de 3 e assim sucessivamente até que o menor numero marcado ao quadrado seja maior que n(número máximo até ao qual pesquisar) e assim no final os números não marcados são primos.

1. Create list of unmarked natural numbers 2, 3, …, $n$
2. $k \leftarrow 2$
3. Repeat
   (a) Mark all multiples of $k$ between $k^2$ and $n$
   (b) $k \leftarrow$ smallest unmarked number $> k$
   until $k^2 > n$
4. The unmarked numbers are primes

Figura 1: Pseudocódigo do algoritmo Sieve of Eratosthenes

## 3.1 Implementação Sequencial

```
1  void sieveSequential(bool *numbers, long long n) {
2      long long k = 2;
3
4      for (long long i = 0; i <= n; i++) {
5          numbers[i] = false;
6      }
7      do {
8          markMultiples(k, numbers, n);
9          k = getSmallestUnmarkedOver(k, numbers, n);
10     } while (k*k <= n);
11 }
```

Nesta primeira implementação sequencial o primeiro ciclo for serve para inicializar o array dos números naturais até n com false. Depois temos um ciclo while que vai correr enquanto

$k^2$ for menor ou igual a N.

```
1  void markMultiples(long long k, bool *numbers, long long n) {
2      long long lowerBound = k * k;
3      for (long long i = lowerBound; i <= n; i += k) {
4          numbers[i] = true;
5      }
6  }
```

```
1  long long getSmallestUnmarkedOver(long long k, bool *numbers, long long n) {
2      for (long long i = k + 1; i <= n; i++) {
3          if (!numbers[i]) {
4              return i;
5          }
6      }
7      cout << "getSmallestUnmarkedOver(): This shouldn't happen" << endl;
8      return LLONG_MIN;
9  }
```

A função *markMultiples* marca todos os múltiplos de k até N e depois a função *getSmallestUnmarkedOver* retorna o número mais baixo por marcar que irá ser o novo k para a iteração seguinte.

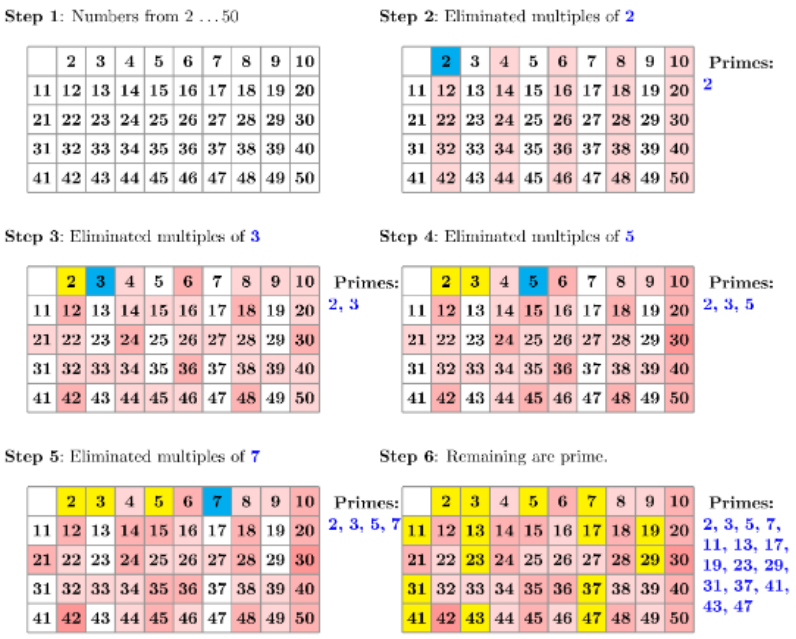


Figura 2: Funcionamento da Implementação Sequencial com N = 50

## 3.2 Implementação Paralela - OpenMP

Uma primeira estratégia de paralelização do algoritmo usa a tecnologia OpenMP, baseada em *threads* e sendo portanto uma tecnologia de memória partilhada. Através de simples diretivas *#pragma*, o OpenMP gere toda a lógica de *threads* necessária.

Não foi possível paralelizar o algoritmo por completo, dado que a descoberta do $k$ de cada iteração é indeterminista. Assim, apenas 2 partes do algoritmo foram paralelizadas: a inicialização do array *numbers* (linhas 4-7), e a marcação dos números múltiplos de $k$

(linhas 11-14). O que requeriria uma alteração significativa ao código sequencial torna-se simplesmente, graças ao OpenMP, em duas novas diretivas *#pragma omp parallel for* nas linhas 4 e 11. Apesar da impossibilidade de paralelização completa, os ganhos no desempenho obtido são já significativos, como será visto na secção de resultados e análise.

```
1  void sieveParallel(bool* numbers, long long n){
2      long long k = 2;
3
4      #pragma omp parallel for
5      for (long long i = 0; i <= n; i++) {
6          numbers[i] = false;
7      }
8
9      do {
10         long long lowerBound = k * k;
11         #pragma omp parallel for
12         for (long long i = lowerBound; i <= n; i += k) {
13             numbers[i] = true;
14         }
15         k = getSmallestUnmarkedOver(k, numbers, n);
16     } while (k*k <= n);
17 }
```

## 3.3 Implementação Paralela - MPI

Outra abordagem é usando o MPI, uma tecnologia de memória distribuída, ou seja, usa processos ao invés de *threads*. Cada processo tem a sua própria memória, tendo cada um uma porção da lista de números a pesquisar. Isto possibilita então a que o algoritmo seja distribuído por diferentes máquinas.

Cada processo cria a sua porção da lista de números tendo em conta o seu *rank* face aos outros processos, recorrendo às macros BLOCK_SIZE, BLOCK_LOW e BLOCK_HIGH (linhas 16-18). De seguida, cada processo executa o algoritmo normalmente, marcando os múltiplos na sua porção da lista, mas é apenas o processo *root* (processo com o id 0) que descobre o próximo valor de $k$ e o envia para todos os processos com a chamada a *MPI_Bcast()* (linha 49). É então o processo *root* o responsável por avançar na iteração do algoritmo. De certa forma, a estratégia de paralelização é a mesma que a usada anteriormente com OpenMP no sentido em que cada processo/*thread* marca os não-primos da sua porção da lista de números (neste caso, linhas 39-40).

No final, cada processo conta os primos da sua porção da lista e envia a contagem ao processo *root* na chamada a *MPI_Reduce()*, que a soma à contagem final.

```
1  void sieveDistributed(int power) {
2      unsigned long long n = pow(2, power);
3
4      MPI_Init(NULL, NULL);
5
6      int numProcesses;
7      int processRank;
8      bool *list;
9      unsigned long long startBlockValue = LLONG_MIN , counter = 0, primes = 0;
10     double openMPITime = 0;
11
12     MPI_Comm_rank(MPI_COMM_WORLD, &processRank);
13     MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
14
15     // calculate each block's boundaries
16     unsigned long long blockSize = BLOCK_SIZE((unsigned long long) processRank
           , (unsigned long long) (n - 1), (unsigned long long) numProcesses);
17     unsigned long long lowValue = BLOCK_LOW((unsigned long long) processRank,
           (unsigned long long) (n - 1), (unsigned long long) numProcesses) + 2;
```

```
18      unsigned long long highValue = BLOCK_HIGH((unsigned long long) processRank
            , (unsigned long long) (n - 1), (unsigned long long) numProcesses) +
            2;
19
20      list = newList(blockSize);
21
22      MPI_Barrier(MPI_COMM_WORLD);
23
24      if (processRank == 0) {
25          openMPITime = -MPI_Wtime();
26      }
27
28      for (unsigned long long k = 2; k*k <= n;) {
29          // calculate the start block value to each process
30          if (pow(k, 2) < lowValue) {
31              lowValue% k == 0 ?
32                  startBlockValue = lowValue :
33                  startBlockValue = lowValue + (k - (lowValue % k));
34          } else {
35              startBlockValue = pow(k, 2);
36          }
37
38          // mark multiples
39          for (unsigned long long i = startBlockValue; i <= highValue; i += k)
40              list[i - lowValue] = NOT_PRIME;
41
42          // get the next prime to broadcast it to the other processes
43          if (processRank == 0) {
44              do {
45                  k++;
46              } while (list[k - lowValue] == NOT_PRIME && pow(k, 2) < highValue)
                    ;
47          }
48
49          MPI_Bcast(&k, 1, MPI_LONG, 0, MPI_COMM_WORLD);
50      }
51
52      if (processRank == 0) {
53          openMPITime += MPI_Wtime();
54          cout << "Time: " << openMPITime << "s\n";
55      }
56
57      for (unsigned long long i = 0; i < blockSize; i++) {
58          if (list[i] == PRIME) {
59              counter++;
60              //cout << i + lowValue << " is prime\n";
61          }
62      }
63
64      if (numProcesses > 1) {
65          MPI_Reduce(&counter, &primes, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD)
                ;
66      } else {
67          primes = counter;
68      }
69
70      if (processRank == 0) {
71          cout << "Number of primes: " << primes << endl;
72      }
73
74      MPI_Finalize();
75  }
```

### 3.4 Implementação Paralela - MPI com OpenMP

É ainda possível combinar as duas tecnologias anteriores para ter o melhor dos dois mundos. OpenMP usa um modelo de memória partilhada para paralelizar *threads* dentro da mesma máquina. MPI usa um modelo de memória distribuída, usando processos e permitindo então distribuir o trabalho por diferentes máquinas. Assim, é possível usar MPI para paralelizar diferentes máquinas com apenas 1 processo em cada e usar OpenMP para paralelizar dentro de cada uma.

Com esta perspetiva, é fácil introduzir OpenMP na versão anterior de MPI - basta introduzir uma diretiva *#pragma omp parallel for* no ciclo que marca os não-primos para cada valor de *k*.

```
1         #pragma omp parallel for
2         for (unsigned long long i = startBlockValue; i <= highValue; i += k) {
3             list[i - lowValue] = NOT_PRIME;
4         }
```

O OpenMP também pode ser usado para contar os primos em cada máquina, após o algoritmo os calcular. Basta, mais uma vez, introduzir uma diretiva *#pragma omp parallel for* mas desta vez usando uma *reduction* para somar o número de primos obtidos em cada *thread*.

```
1     #pragma omp parallel for reduction(+:counter)
2     for (unsigned long long i = 0; i < blockSize; i++) {
3         if (list[i] == PRIME) {
4             counter++;
5             //cout << i + lowValue << " is prime\n";
6         }
7     }
```

## 4 Metodologia de análise

Todos os algoritmos foram implementados em C++. Foram medidos os tempos para inputs de $2^{25}$, $2^{28}$ e $2^{32}$, ou seja, 33 554 432, 268 435 456 e 4 294 967 296 respetivamente, para as seguintes configurações. Estas medidas foram realizadas graças à função *omp_ get_ wtime()* para as versões sequencial e paralela com OpenMP, e a função *MPI_ Wtime()* para a versão paralela com MPI.

Obtendo os tempos, é possível avaliar o desempenho calculando o *speedup* (S) e a *eficiência* (E) para cada configuração.

O *speedup* representa a melhoria no tempo de execução de qualquer das versões paralelas do algoritmo em função do tempo da versão sequencial, e é dado pela equação:

$$Speedup = \frac{T_{seq}}{T_{parallel}}$$

A *eficiência* representa o aproveitamento dos processadores (P). Com esta estatística conseguimos dizer que uma certa versão do algoritmo é escalável caso se encontre no intervalo $E \in [0, 1]$.

$$E = \frac{Speedup}{P}$$

Todos os testes foram realizados no mesmo tipo de máquina:

- CPU: i7-4790 (3.6/4.0GHz, 8MB L3 cache)
- RAM: 24GB DDR3
- OS: Ubuntu 18.04 LTS

# 5 Resultados e análise

## 5.1 Sequencial

Na seguinte tabela estão indicados os tempos medidos para a versão sequencial deste algoritmo. Nota-se que o tempo decorrido é proporcional à dimensão dos dados, dado que o fator de 25 para 28 é cerca de 9 (próximo do aumento de 8 vezes na dimensão) e o fator de 28 para 32 é cerca de 18 (próximo do aumento de 16 vezes na dimensão).

| **Expoente (2^N)** | 25 | 28 | 32 |
|---|---|---|---|
| **Tempo (s)** | 0,517 | 4,670 | 85,182 |

## 5.2 Paralela - OpenMP

As colunas dos gráficos abaixo referem-se ao número de *threads* fornecidas ao algoritmo - de 1 a 8.

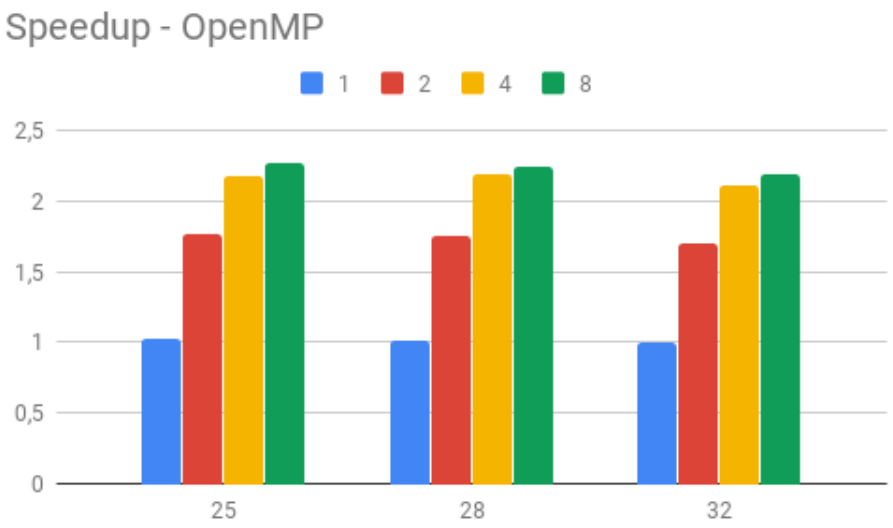


Figura 3

O *speedup* aumenta à medida que são usadas mais threads. Num caso utópico, o *speedup* resultante seria igual ao número de *threads* e assim teríamos um algoritmo perfeitamente paralelo em que duas threads resultavam num desempenho duas vezes melhor relativamente a uma thread. Porém, tal é praticamente impossível no geral. Neste caso, o *speedup* não é ideal mas, felizmente, aumenta quando são fornecidas mais *threads*, resultando num desempenho melhor. Os maiores ganhos verificam-se quando se passa de 1 para 2 *threads*, e ainda há um ganho decente para 3 *threads*, mas dificilmente se justificar usar 4.

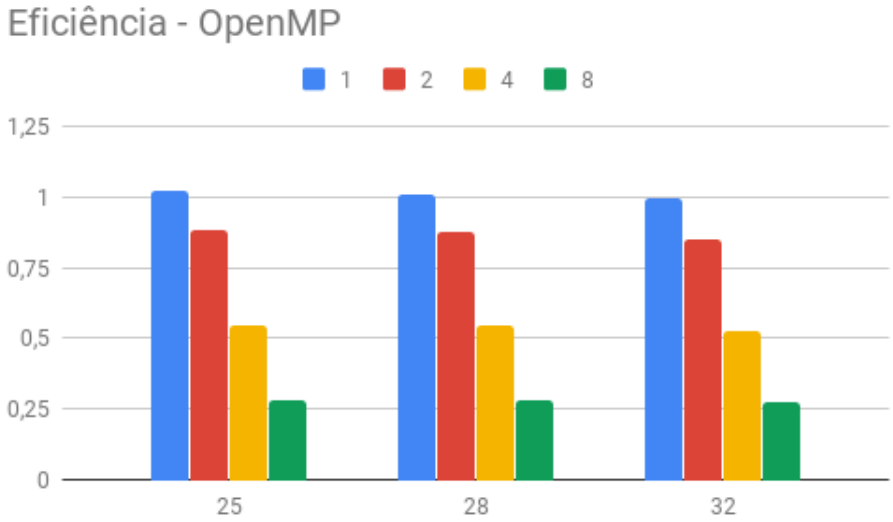


Figura 4

De acordo com a definição da eficiência previamente estabelecida, esta versão do algoritmo é escalável pois os valores mantêm-se todos entre 0 e 1 à medida que se aumenta o intervalo de entrada e o número de threads utilizadas.

## 5.3 Paralela - MPI

As colunas dos gráficos abaixo referem-se ao número de processos usados pelo algoritmo, no formato *PC's * processos*. Assim, o algoritmo foi corrido em 1 a 4 PC's, usando 2 processos em cada. No entanto, o algoritmo foi executado em PC's da FEUP e, infelizmente, a natureza instável da rede resultou em tempos de execução fracos e severamente inconsistentes, pelo que não foi possível retirar conclusões significativas desta versão.

Ainda assim, verifica-se um *speedup* superior a 1 associado a todas as configurações, o que demonstra a importância da paralelização apesar das condições árduas do ambiente.

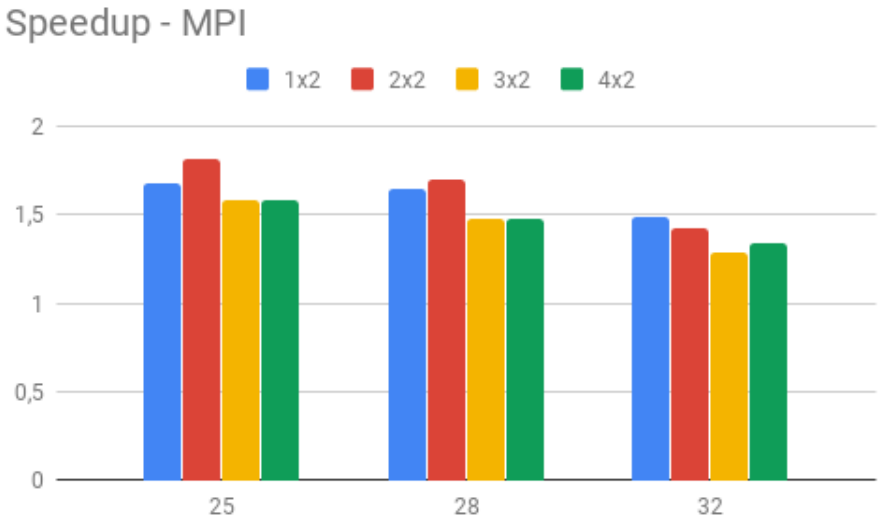


Figura 5

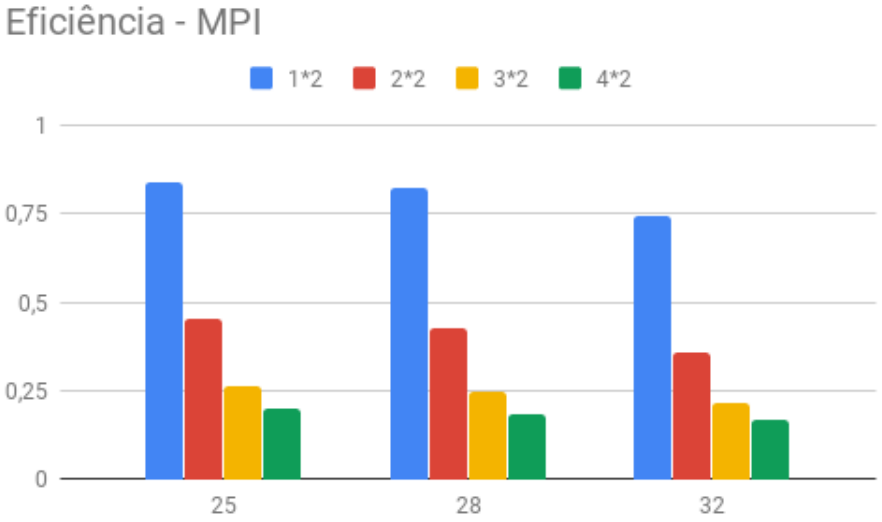


Figura 6

### 5.4 Paralela - MPI com OpenMP

Esta é a versão que melhor resultados obteve. Com MPI foram testados 1 a 4 PC's e em cada um correram 8 *threads* graças ao OpenMP.

O *speedup* teve melhorias muito significativas, sendo esta a versão mais escalável até agora apesar da eficiência ser menor no geral. Isto é porque a eficiência é medida em função do elevado número de *threads*, a que é inversamente proporcional, mas mais importante que isso é o facto de a eficiência não só manter-se praticamente constante em alguns casos que se adicionaram *threads*, como aumentou significativamente noutros. A eficiência manteve-se

entre 0 e 1 e portanto, mantendo a definição referida numa secção anterior, esta versão do algoritmo pode ser considerada escalável.

Porém, a instabilidade da rede da FEUP continua a ser um factor. À medida que a dimensão dos dados aumenta, verificam-se melhorias mais pequenas e chegam-se mesmo a registar piorias no caso de maior dimensão dos dados. É de notar que estes últimos foram também algo inconsistentes. No entanto, como desta vez apenas há um processo em cada PC, graças à paralelização interna do OpenMP, e assim existem menos mensagens a serem trocadas entre os PC's, a rede não afeta tanto os resultados e as melhorias resultantes desta abordagem paralela são bem visíveis.

Julgamos que o problema dos dados de maior dimensão se deva a haver uma maior probabilidade de ocorrerem falhas na rede, dado que o tempo de cálculo necessário é maior.

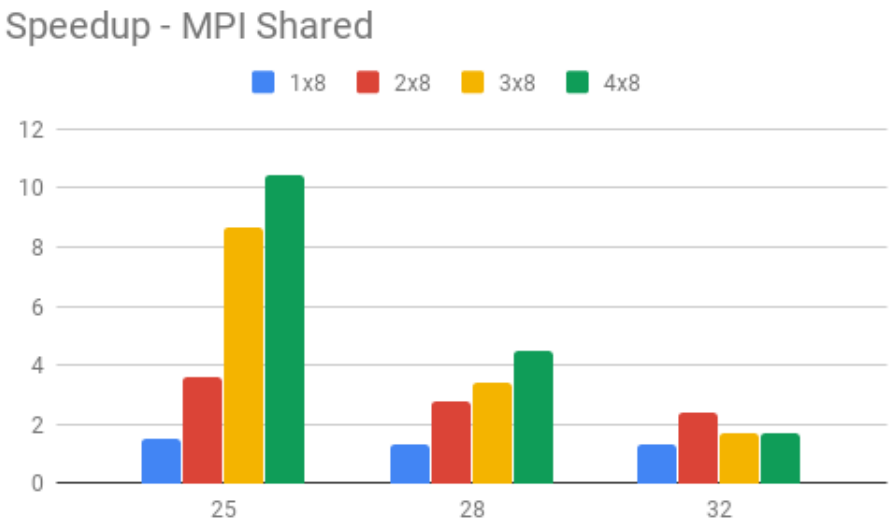


Figura 7

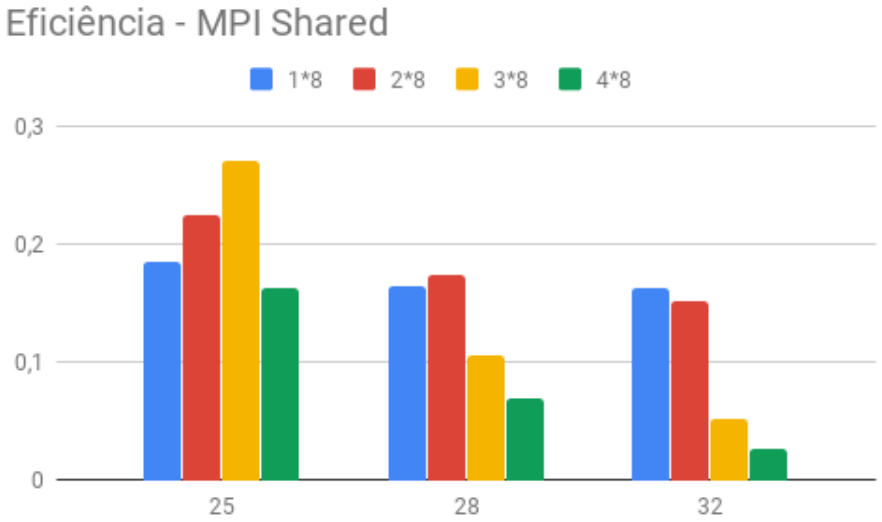


Figura 8

# 6 Conclusão

Neste trabalho tivemos a oportunidade de usar pela primeira vez diferentes API's de computação paralela, OpenMP e MPI, aplicando os conhecimentos das aulas.

Desenvolvemos 3 diferentes abordagens com estas: OpenMP, MPI e uma mistura de ambos. OpenMP revela ser o ideal dentro de um único PC, mas para usar múltiplas máquinas é requerido usar MPI. Contudo, a comunicação entre PC's está dependente da qualidade da infraestrutura, pelo que uma maneira de mitigar este factor será o de combinar o MPI com o OpenMP, usando as *threads* do último em cada PC ao invés de múltiplos processos. Esta última abordagem revelou até ser a mais escalável nas nossas experiências.

# Referências

[1] Open MPI v4.0.1 documentation
`https://www.open-mpi.org/doc/current/`