Paralelização do Algoritmo *Sieve of Eratosthenes*

Universidade do Porto

Faculdade de Engenharia

FEUP

Computação Paralela

4^o **ano** 2^o **semestre**

Mestrado Integrado em Engenharia Informática e Computação

Hugo Drumond	201102900	hugo.drumond@fe.up.pt
Pedro Moura	201306843	up201306843@fe.up.pt

1 Introdução

No âmbito da Unidade Curricular de computação paralela foi proposto fazer um estudo sobre a paralelização do algoritmo *Sieve of Eratosthenes* através do uso de ferramentas que auxiliam o desenvolvimento de programas com memória partilhada e memória distribuída, respectivamente *OpenMP* e *OpenMPI*. Assim sendo, é feita a descrição do algoritmo *The Sieve of Eratosthenes*, e explicados os melhoramentos nas nossas implementações. Além disso, é demonstrado o resultado de várias experiências, a análise de cada qual, e as conclusões a que se chegou.

2 Descrição do Problema

Sieve of Eratosthenes é um algoritmo que serve para encontrar todos os números primos até um dado limite. O objetivo deste trabalho é o estudo da sua paralelização de forma a torná-lo mais escalável.

3 Algoritmo *The Sieve of Eratosthenes*

Querendo achar todos os números primos num intervalo [2,n], cria-se uma lista de todos os números inteiros consecutivos nesse intervalo. Onde posteriormente serão marcados todos os múltiplos dos números primos. Até ao primo cujo valor seja menor ou igual à raiz quadrada do último número da lista.

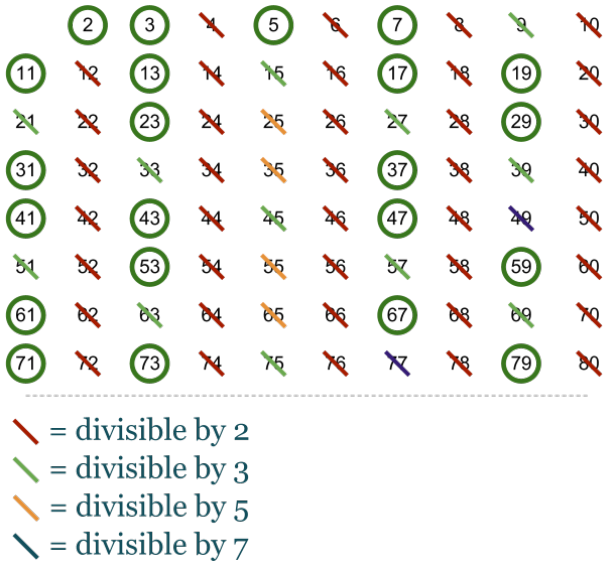

Figura 1: Exemplo do funcionamento do algoritmo

A marcação desses números começa pela seleção do 1º número da lista, o 2. Após essa seleção são marcados todos os seus múltiplos, a partir do multiplicador com valor igual ao número selecionado. Ou seja, marca-se o 4, 6, 8, 10, etc até ao último número da lista. Quando esta última condição se verificar será selecionado o próximo número da lista que não se encontre marcado, que neste caso será o 3. E assim por diante, até ao número primo que seja menor ou igual a sqrt(último número da lista). Tal acontece para que os múltiplos do primo em questão não ultrapassem o limite.

Os números primos serão os valores da lista que não foram marcados. De facto, o modo de funcionamento do algoritmo é semelhante ao uso de múltiplas peneiras (marcação dos múltiplos dos primos), daí o uso da palavra *sieve* no algoritmo que em português tem o significado de peneira.

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 2: Pseudocódigo das aulas

Será também importante referir que a complexidade espacial e temporal é, respetivamente, O(n) e O(n log log n).

4 Implementação e melhoramento do Algoritmo

Para a implementação deste algoritmo usou-se a linguagem *C++*, com o auxílio das bibliotecas *OpenMP* e *OpenMPI*. Cada implementação foi colocada num projeto à parte de modo a tornar o código o mais legível possível. Deste modo, foram criados os seguintes projetos:

- i-sequential: *sequential, single machine, single core*
- i-sequential-improved: *sequential, single machine, single core, with code optimizations*
- ii-openmp: *single machine, multiple threads*
- ii-openmp-improved: *single machine, multiple threads, with code optimizations*
- iii-mpi: *multiple machines, multiple processes per machine*
- iii-mpi-improved: *multiple machines, multiple processes per machine, with code optimizations,* ***works perfectly but was not benchmarked***
- iv-mpiOpenmp: *multiple machines, single process per machine, multiple threads per machine*
- iv-mpiOpenmp-improved: *multiple machines, single process per machine, multiple threads per machine, with code optimizations,* ***works perfectly but was not benchmarked***

De modo a melhorar a complexidade temporal e espacial foram desprezados os múltiplos de 2 nas implementações **-improved*. Tal pode ser feito usando uma *wheel* [2] ou seja dando saltos de 2. Em todas as implementações é usada uma lista de booleanos inicializada a falso, sendo esse valor passado no construtor *new bool[last_number - 1]{false}*. Esta inversão de lógica é muito mais eficiente visto que o sistema operativo faz *scrubbing* de memória para zeros continuamente. Permitindo assim uma inicialização de memória mais rápida. Um booleano a *true* tem o significado que o número associado a essa posição foi filtrado, ou seja, não é primo. Na versão *sequential improved* e *mpi improved* foi utilizado um *vector<bool>*, visto que internamente é usado um *dynamic_bitset*. Tal permite encaixar mais valores numa *cache line* resultando assim numa diminuição do tempo de execução.

4.1 Algoritmos Sequenciais

4.1.1 *Sequential*

Nesta versão apenas foi feita uma implementação direta do algoritmo *Sieve of Eratosthenes*. Segundo o que está descrito no ponto 3 do relatório. O código encontra-se no ficheiro *i-sequential/src/main.cpp*.

```
bool *sieved_vector = new bool[last_number - 1]{false};
size_t limit = static_cast<size_t>(sqrt(static_cast<double>(last_number)));
for (size_t k = 2; k <= limit;) {
  for (size_t multiple = k * k; multiple <= last_number; multiple += k) {
    sieved_vector[multiple - 2] = true;
  }
  do {
    k += 1;
  } while (k <= limit && sieved_vector[k - 2]);
}
```

4.1.2 *Sequential Improved*

O código encontra-se no ficheiro *i-sequential-improved/src/main.cpp*. É usado um *vector<bool>* e são ignorados os pares como já foi explicado no ponto 4.

```
vector<bool> sieved_vector(last_number / 2, false);
size_t limit = static_cast<size_t>(sqrt(last_number));
for (size_t k = 3; k <= limit; k += 2) {
  if (!sieved_vector[k / 2]) {
    for (size_t multiple = k * k; multiple < last_number; multiple += 2 * k) {
      sieved_vector[multiple / 2] = true;
    }
  }
}
```

4.2 Algoritmos Paralelizados

De forma a aumentar a performance da versão sequencial foram codificadas diferentes versões deste usando técnicas de paralelização. Estas implementações visam dividir o problema em várias partes concorrentes de forma a melhor utilizar os recursos disponíveis. Tal pode ser feito recorrendo a *threads* com memória partilhada e/ou processos em diversas máquinas cada uma com memória local independentemente entre si (distribuída).

Notou-se que a classe *vector* do *C++* não deve ser usada para as versões paralelas *OpenMP* uma vez que causa *race conditions*. Tal acontece porque internamente é usado um *dynamic_bitset* para guardar os booleanos. Que implica ter de carregar um bloco de *bytes*(número de blocos dependente da implementação) só para alterar o valor de um *bit*.

4.2.1 *OpenMP*

Como referido anteriormente a versão de memória partilhada tem como base a divisão de um processo em várias *threads* concorrentes, em que existe partilha de memória entre elas. O código encontra-se no ficheiro *ii-openmp/src/main.cpp*.

```
bool *sieved_vector = new bool[parsed.last_number - 1]{false};
size_t limit =
    static_cast<size_t>(sqrt(static_cast<double>(parsed.last_number)));
for (size_t k = 2; k <= limit;) {
#pragma omp parallel for num_threads(parsed.number_of_threads) schedule(static)
  for (size_t multiple = k * k; multiple <= parsed.last_number;
       multiple += k) {
    sieved_vector[multiple - 2] = true;
  }
  do {
    ++k;
  } while (k <= limit && sieved_vector[k - 2]);
}
```

O código selecionado para paralelização foi o ciclo onde todos os múltiplos dos números primos são marcados. Visto ser o único que é canónico. Este é dividido por *parsed.number_of_threads threads*.

4.2.2 *OpenMP Improved*

Nesta implementação os pares são desprezados e o *pragma* é colocado no ciclo mais exterior de modo a reduzir o *overhead* na criação de *threads*. O código encontra-se no ficheiro *ii-openmp-improved/src/main.cpp*.

```
bool *sieved_vector = new bool[parsed.last_number / 2]{false};
size_t limit = static_cast<size_t>(sqrt(parsed.last_number));
#pragma omp parallel for num_threads(parsed.number_of_threads) schedule(static)
for (size_t k = 3; k <= limit; k += 2) {
  if (!sieved_vector[k / 2]) {
    for (size_t multiple = k * k; multiple < parsed.last_number; multiple += 2 * k) {
      sieved_vector[multiple / 2] = true;
    }
  }
}
```

Não é necessário criar locks visto que:

- O resultado final não é afetado mesmo que uma *thread* escolha um não primo para filtrar. Pois todos os múltiplos desse número não podem ser primos. Tal poderá acontecer devido ao "atraso" da *thread* anterior ou a *race conditions*
- Só é possível marcar um número como não primo

4.2.3 *OpenMPI*

Nesta versão, *iii-mpi/src/main.cpp*, o problema é dividido por várias máquinas que estão interligadas por rede. Cada qual com capacidade para múltiplos processos, normalmente *#processes <= #cores* (*slots* e *max-slots*). Em suma um dado processo será responsável por uma parte do *array*, um bloco. Esta divisão é codificada pelas seguintes *macros*:

```
// taken from the class pdf
#define BLOCK_LOW(i, n, p) ((i)*(n)/(p))
#define BLOCK_HIGH(i, n, p) (BLOCK_LOW((i)+1,n,p)-1)
#define BLOCK_SIZE(i, n, p) (BLOCK_LOW((i)+1,n,p)-BLOCK_LOW(i,n,p))
#define BLOCK_OWNER(index, n, p) ((((p)*(index)+1)-1)/(n))
```

Ao fornecer, *i*, o índice do processo(*rank*), *n*, o número de elementos da lista, e finalmente, *p*, o número de processos; é possível definir a responsabilidade de dados de cada *rank*. *BLOCK_LOW* é o valor mínimo. *BLOCK_HIGH* o valor máximo, que é calculado indo buscar o *BLOCK_LOW* do próximo *rank* e subtraindo uma unidade. E, *BLOCK_SIZE*, o número de elementos do *rank*.

Este modo de divisão por blocos só está pensado para uma gama de valores divisível pelo número de processos e em que o *#elements >= #processes*. Para que fosse mais robusto teríamos de distribuir o resto pelos blocos e garantir que no *edge case* o *#processes* passaria a ser igual ou menor que o *#elements*. Como o *#elements* esperado pelo programa é sempre *2n*, tem-se que, *#processes* também tem de ser uma potência de dois. Ademais é garantido que *#elements* é muito maior que *#processes* visto que o intervalo de teste está compreendido entre o expoente 25 e 32 inclusivé.

Segue a codificação do problema acoplado de algumas anotações.

```
// All processes use the first prime, 2, to filter
size_t k = 2;
size_t startBlock;
while (k * k < limit) {
  // Determine the startBlock for the peers
  if (k * k < blockLow) {
    // If the blockLow value is divisible by k
    // blockLow should be filtered and used to filter
    startBlock = blockLow;
    if (blockLow % k != 0) {
      startBlock += (k - (blockLow % k)); // Else find the closest multiple
    }
  } else {
    startBlock = k * k;
  }

  // Mark the multiples, if startBlock is greater than blockHigh don't mark
  // because it exceeds this rank's concerns
  for (size_t multiple = startBlock; multiple <= blockHigh; multiple += k) {
    sieved_vector[multiple - blockLow] = true;
  }

  // Only root finds the primes because the last prime to be used to filter
  // is in the root block, p < sqrt(n)
  if (rank == ROOT_MACHINE) {
    do {
      k++;
    } while (k * k < blockHigh && sieved_vector[k - blockLow]);
  }
```

```
  // Root sends another prime to all members of the MPI_COMM_WORLD
  // communication group so that they can begin filtering again
  MPI_Bcast(&k, 1, MPI_UNSIGNED, ROOT_MACHINE, MPI_COMM_WORLD);
}
```

4.2.4 *OpenMPI Improved*

Esta versão, *iii-mpi-improved/src/main.cpp*, elimina os pares e usa um *vector<bool>* para aumentar os *cache hits* como já foi explicado anteriormente.

```
size_t number_odd_elements = last_number / 2;
unsigned long blockSize = BLOCK_SIZE(rank, number_odd_elements, size);
unsigned long blockLow = BLOCK_LOW(rank, number_odd_elements, size) * 2 + 1;
unsigned long blockHigh = BLOCK_HIGH(rank, number_odd_elements, size) * 2 + 1;
vector<bool> sieved_vector(blockSize, false);

MPI_Barrier(MPI_COMM_WORLD);
if (rank == ROOT_MACHINE) {
  start = MPI_Wtime();
}

size_t startBlock;
for (size_t k = 3; k <= static_cast<size_t>(sqrt(last_number));) {
  if (k * k < blockLow) {
    startBlock = blockLow;
    if (blockLow % k != 0) {
      startBlock += -(blockLow % k) + k;
      if (startBlock % 2 == 0) {
        startBlock += k;
      }
    }
  } else {
    startBlock = k * k;
  }

  for (size_t multiple = startBlock; multiple <= blockHigh; multiple += 2 * k) {
    sieved_vector[(multiple - blockLow) / 2] = true;
  }

  if (rank == ROOT_MACHINE) {
    k += 2;
  }
  MPI_Bcast(&k, 1, MPI_UNSIGNED, ROOT_MACHINE, MPI_COMM_WORLD);
}
```

4.2.5 *OpenMPI-OpenMP*

Nesta implementação, *iv-mpiOpenmp/src/main.cpp*, a versão OpenMP e OpenMPI foram conjugadas de modo a:

- usufruir do potencial de várias máquinas
- diminuir o *overhead* de ter vários processos numa mesma máquina

Esta paralelização híbrida usa tanto memória distribuída como partilhada. E tira partido de *threads*. A configuração ideal para esta implementação seria, um processo por nó cada qual com o número de *threads* recomendado nessa máquina.

```
size_t k = 2;
size_t startBlock;
while (k * k < limit) {
  if (k * k < blockLow) {
    startBlock = blockLow;
    if (blockLow % k != 0)
      startBlock += (k - (blockLow % k));
  } else {
    startBlock = k * k;
  }

#pragma omp parallel for num_threads(parsed.number_of_threads) schedule(static)
  for (size_t multiple = startBlock; multiple <= blockHigh; multiple += k) {
    sieved_vector[multiple - blockLow] = true;
  }

  if (rank == ROOT_MACHINE) {
    do {
      k++;
    } while (k * k < blockHigh && sieved_vector[k - blockLow]);
  }

  MPI_Bcast(&k, 1, MPI_UNSIGNED, ROOT_MACHINE, MPI_COMM_WORLD);
}
```

4.2.6 OpenMPI-OpenMP Improved

Difere da anterior somente por eliminar os pares. Que é semelhante ao que já foi mostrado na versão *OpenMPI Improved*. Código em *iv-mpiOpenmp-improved/src/main.cpp*.

5 Experiências e Análise de Resultados

5.1 Descrição das Experiências

Para cada uma as versões foi desenvolvido um *makefile* com dois *targets*: *debug* e *release*. Quando um projeto é compilado em modo *release* só é impresso o tempo de execução. Caso seja necessário ver os números primos é necessário compilar em modo *debug*. Foram ainda produzidos *scripts* de *benchmark* que efetuam inúmeros testes, cada qual 6 vezes sendo escolhido o menor tempo de execução. Tal foi feito de modo a minimizar o impacto de outros processos que estivessem a correr concorrentemente ou de latências ocasionais na rede. Estes resultados são colocados automaticamente em ficheiros *csv* para fácil importação para ferramentas de análise.

Todas as experiências foram realizadas tendo como base intervalos muito grandes, de expoente 25 até 32 de base 2. Foram usados 4 computadores para efetuar os testes. Com diferentes configurações dependendo do teste em questão.

Experiências efetuadas:

- *Sequential* e *Sequential Improved*

$$2^n, n \in [25, 32]$$

- *OpenMP* e *OpenMP Improved*

$$2^n * n\_threads, n \in [25, 32], n\_threads \in [1, 4]$$

- *OpenMPI*

$$2^n * n\_computers * n\_processes, n \in [25, 32], n\_processes \in \{1, 2, 4\}, n\_computers \in \{1, 2, 4\}$$

- *OpenMPI-OpenMP*

$$2^n * n\_computers * n\_threads, n \in [25, 32], n\_computers \in \{1, 2, 4\}, n\_threads \in [1, 4]$$

As características dos computadores(todos iguais) que produziram os documentos da pasta "benchmarks9-10May" usados no excel são as seguintes: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 4 cores, 2 threads por core, CPU max 4000 MHz, CPU min 800 MHz, L1d cache 32K, L1i cache 32K, L2 cache 256k, L3 cache 8192K, L1d e L1i e L2 por core, L3 shared, e 16gb ddr3 de ram. A análise e os gráficos apresentados foram baseados nestes testes.

5.2 Metodologia de Avaliação

Foram usadas as seguintes métricas de avaliação de desempenho:

$$speedup = \frac{T_{sequential}}{T_{other}}$$

$$instructions/s = \frac{n\ log\ log(n)}{T}$$

5.3 Análise dos Resultados

5.3.1 *Sequential*

Esta versão será usada como referência para o estudo das outras visto que é a implementação mais próxima da definição. No gráfico 3, verifica-se um acentuado decréscimo no número de operações por segundo à medida que a lista de números primos a processar aumenta. Tal indica um acréscimo do número de operações em memória, não só devido ao tamanho da lista dos números onde se vai efetuar as marcações dos múltiplos, mas também ao número de operações e tempo despendido a aceder a esta.

Também será importante referir que a versão sequencial melhorada (que exclui os números pares e usa um *vector*) chega a ter um desempenho praticamente idêntico senão melhor que a versão que obtém o melhor desempenho. A que usa múltiplos computadores e *threads*, *OpenMPI-OpenMP*.

5.3.2 Paralelismo com Memoria Partilhada

No gráfico 4, podemos verificar que houve um grande aumento de desempenho quando se passa de 1 para 2 *threads*. Havendo depois uma pequena melhoria com 3 *threads*.

O desempenho com 4 *threads* é algumas vezes inferior ou idêntico ao de 3. Não apresenta melhores resultados muito provavelmente devido ao *overhead* criado pela abertura de múltiplas *threads*. Não foram apresentados resultados para um número de *threads* superior ao número de cores físicos do *CPU* devido não só à análise feita no trabalho anterior de que o melhor desempenho é geralmente atingido quando as *threads* não ultrapassam as cores da máquina, mas também pelo desempenho de 3 *threads* ser superior ao de 4 *threads* nos nossos testes.

Na versão *Improved* pode-se verificar um melhor desempenho. Contudo não chega a ser melhor que a versão sequencial melhorada, podendo isto ser explicado pelo facto de a versão sequencial usar um *vector<bool>* que tem uma representação interna semelhante a um *dynamic_bitset*. Para que o *dynamic_bitset* pudesse ser usada na versão *OpenMP* seria necessário distribuir os dados pelas *threads* em blocos que fossem múltiplos da representação interna que guarda um conjunto de *bits*. Por outras palavras, para garantir que não há *race conditions*, se o *biset* alocasse inteiros(32 *bits*) as *threads* não poderiam estar responsáveis por partes de um desses blocos. Outra maneira seria utilizar *locks* mas prejudicaria ainda mais o desempenho.

5.3.3 Paralelismo com Memoria Distribuída

Verificou-se que o teste com mais sucesso foi para 2 processos por máquina.

A performance do teste com 2 computadores, 2 processos por computador (4 processos) é muito superior à versão *OpenMP* com 4 ou 3 *threads*. Constata-se até que a versão *OpenMPI* é mais vantajosa que a *OpenMP* quando com um número de máquinas superior ou igual a 2.

Podendo isto ser explicado pelo aumento das instruções/segundo (resultado da contribuição das várias máquinas), quando comparado com a versão *OpenMP*. O aumento das instruções/segundo por máquina deve-se à redução do espaço e tempo gasta em acessos a memória.

5.3.4 Paralelismo com Memória Partilhada e Distribuída

Não se obteve uma melhoria muito acentuada face à versão *OpenMPI* como podemos ver comparando os gráficos 7 e 8. O melhor desempenho terá sido o de 4 máquinas e 3 *threads*, ou seja juntando os dois melhores testes do uso de memória partilhada e distribuída como esperado. Esta é a melhor versão se não contarmos com as otimizações feitas aos algoritmos. Pois tendo isso em conta, das versões *Improved* testadas, a versão *Sequential Improved* chega a estar ao mesmo nível usando 4 vezes menos computadores e 3 vezes menos cores.

6 Conclusões

Em suma concluímos que o uso de memória distribuída acaba por ser mais eficaz em termos de performance que o modelo de memória partilhada e que é preferível usar várias máquinas de menor performance e de custo mais barato que uma mais dispendiosa de múltiplos cores e memória partilhada entre si.

Também concluímos que de todas as versões *benchmarked* (todas exceto *OpenMPI Improved* e *OpenMP-OpenMP Improved*) a versão sequencial melhorada chega a ser melhor que todas as outras, o que nos faz ver que a resposta nem sempre é usar mais recursos mas muitas vezes explorar melhor o problema em causa.

Como continuação do trabalho gostaríamos de efetuar os *benchmarks* restantes e implementar outras versões *Improved*. Explorando: *wheels* maiores, suporte para gamas de números e processos sem restrições de valor (questão da divisão por blocos na versão *OpenMPI*), testes num serviço de *Cloud Computing*, uso de estruturas eficientes que melhorem os *cache hits* mas que não usem *locks*.

7 Referências

- Slides das aulas teóricas
- `https://programmingpraxis.com/2012/01/06/pritchards-wheel-sieve/`
- `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`
- `http://stackoverflow.com/questions/17794569/why-is-vectorbool-not-a-stl-container`

8 Anexos

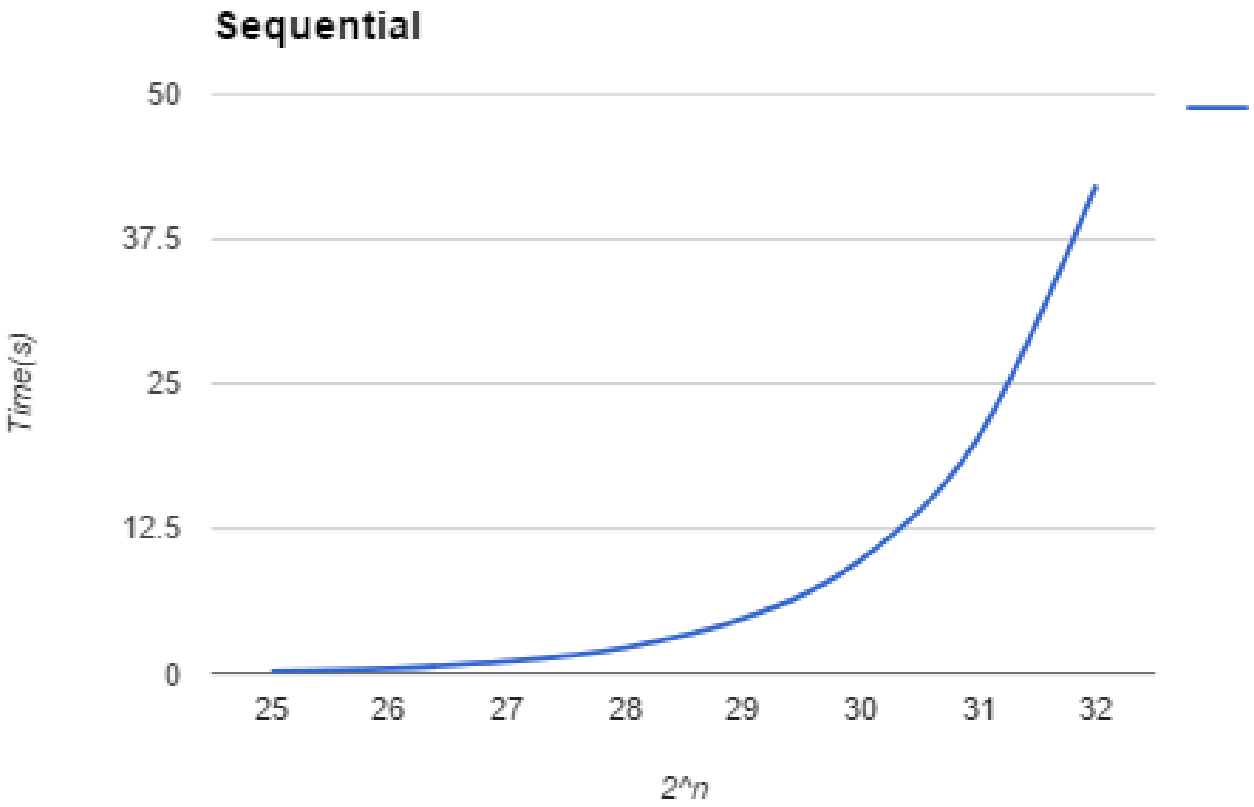

Figura 3: GOP/s -> performance do algoritmo sequencial base

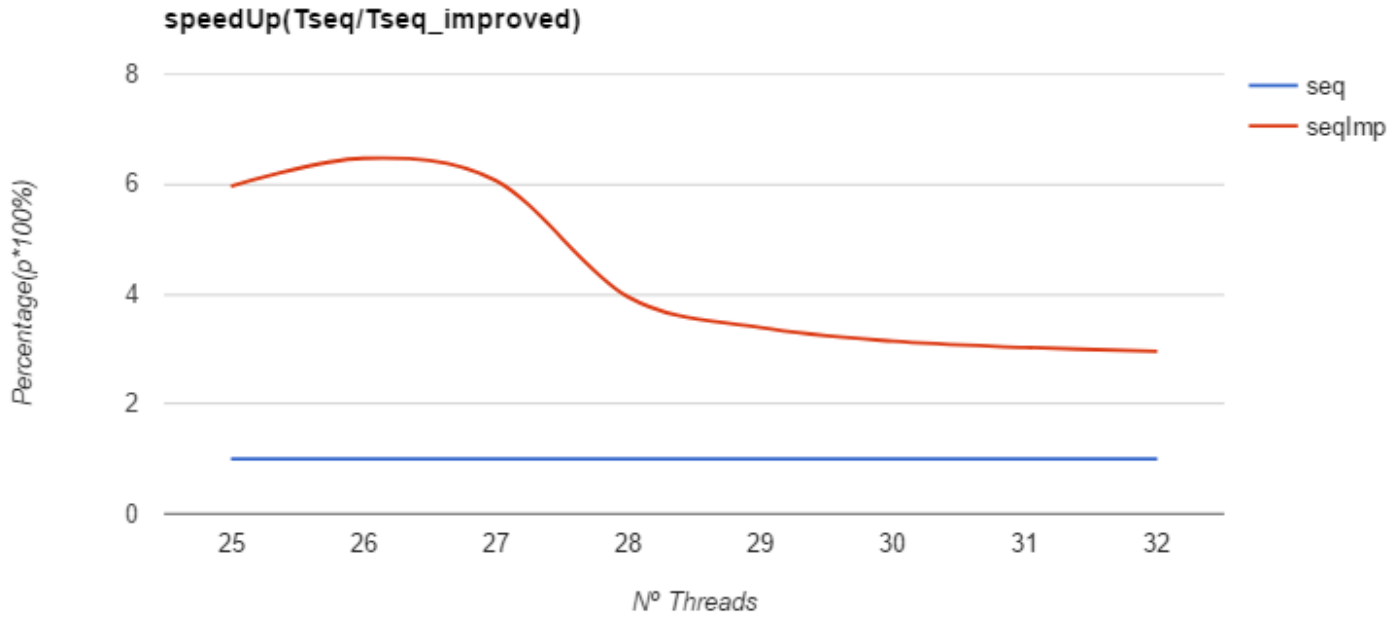

Figura 4: GOP/s -> performance do algoritmo sequencial base e melhorada

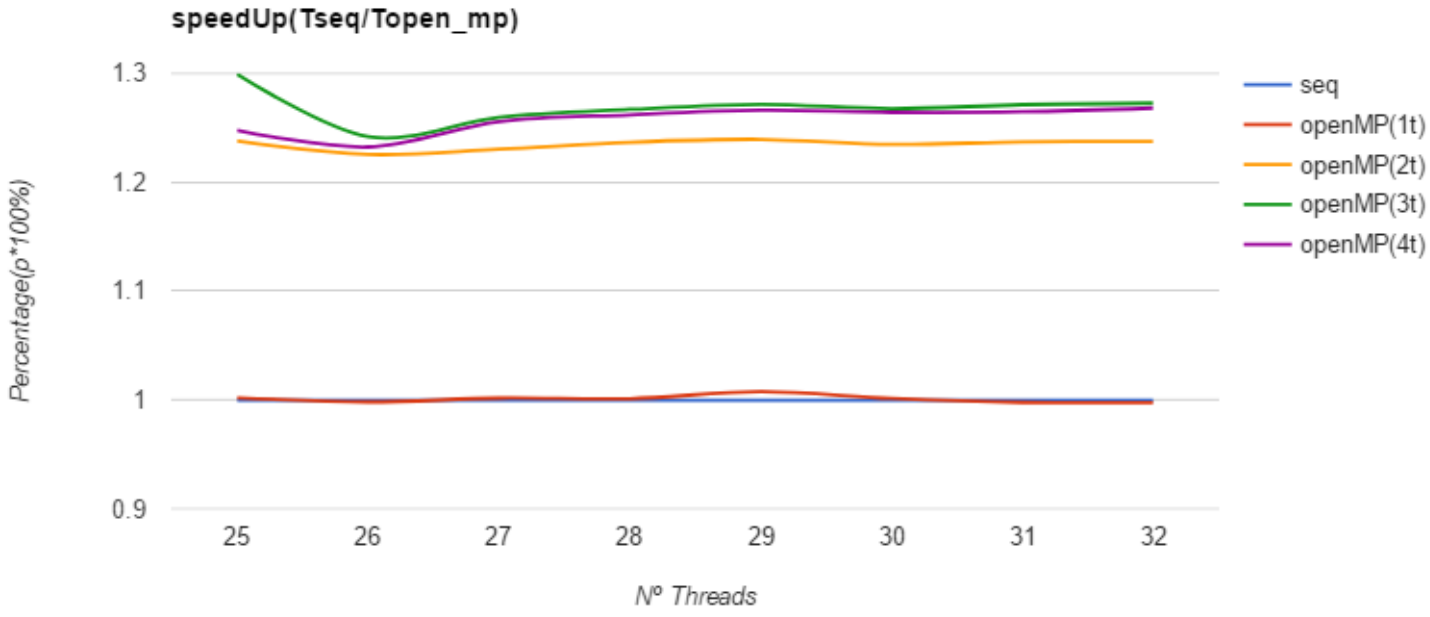

Figura 5: SpeedUp -> da versão usando openMP variando o numero de threads

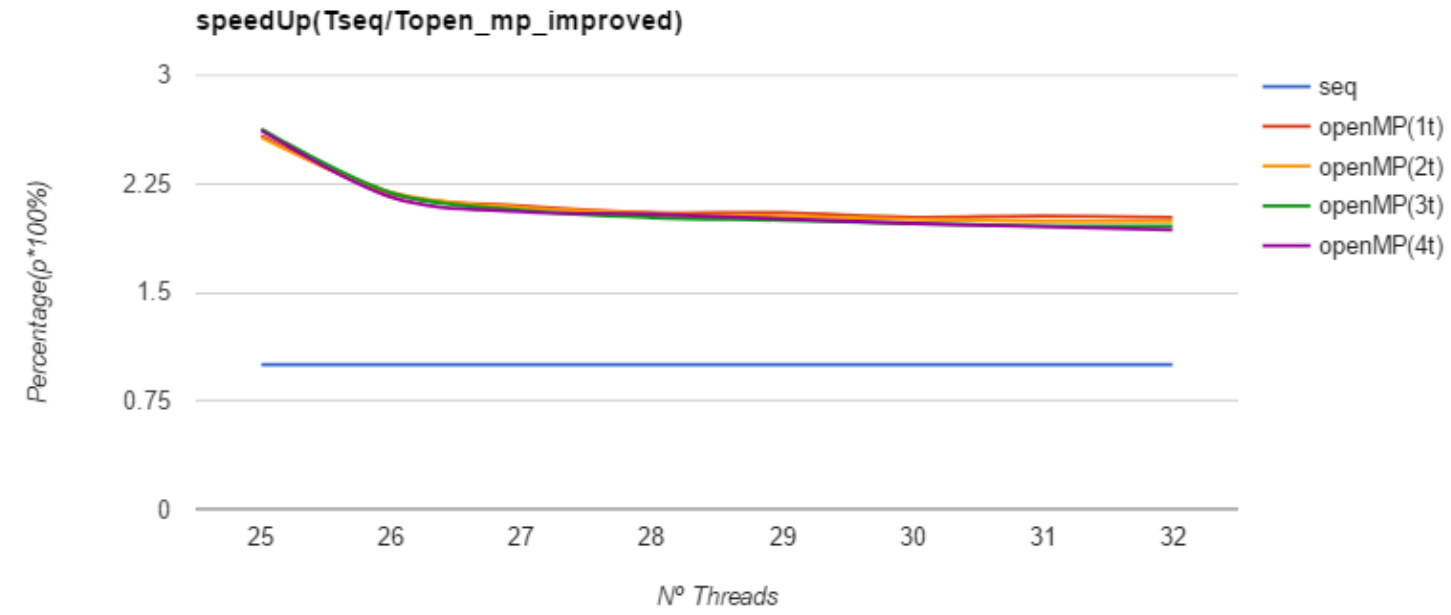

Figura 6: SpeedUp -> da versão usando openMP melhorada variando o número de threads

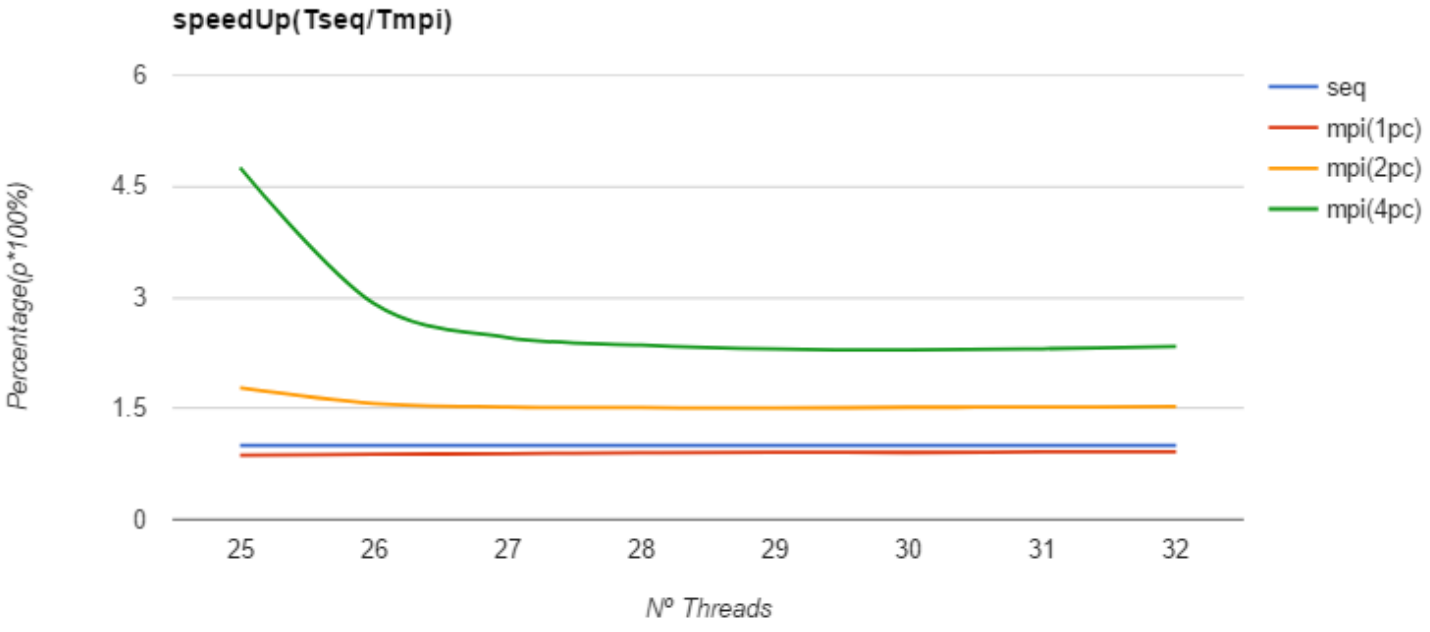

Figura 7: SpeedUp -> da versão usando openMPI variando o numero de processos.

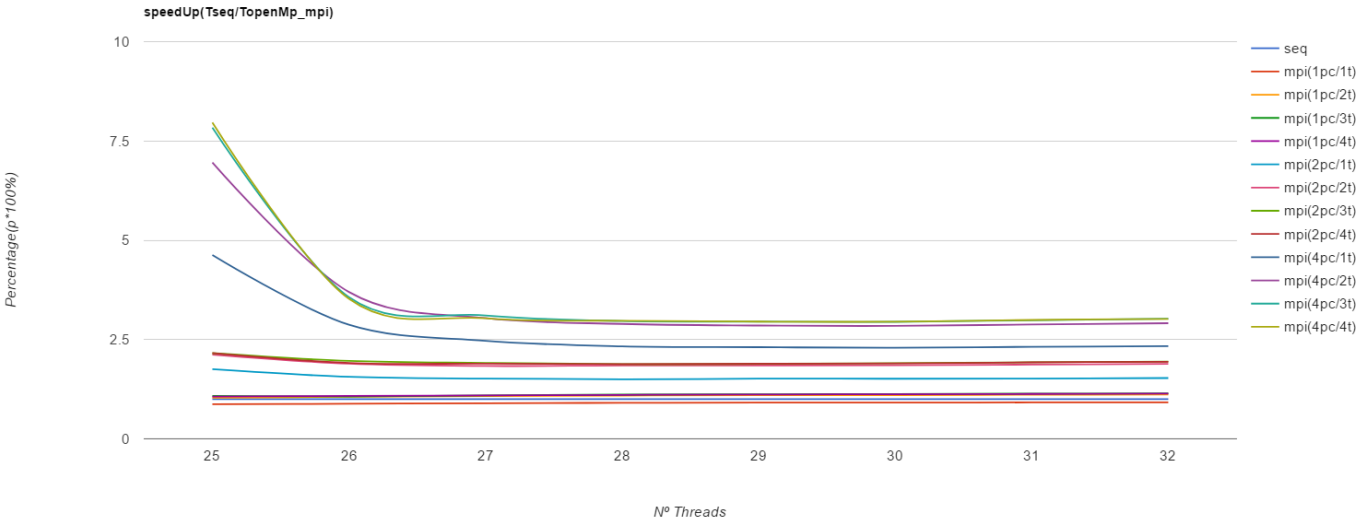

Figura 8: SpeedUp -> da versão usando openMPI + openMP variando o numero de processos e threads.

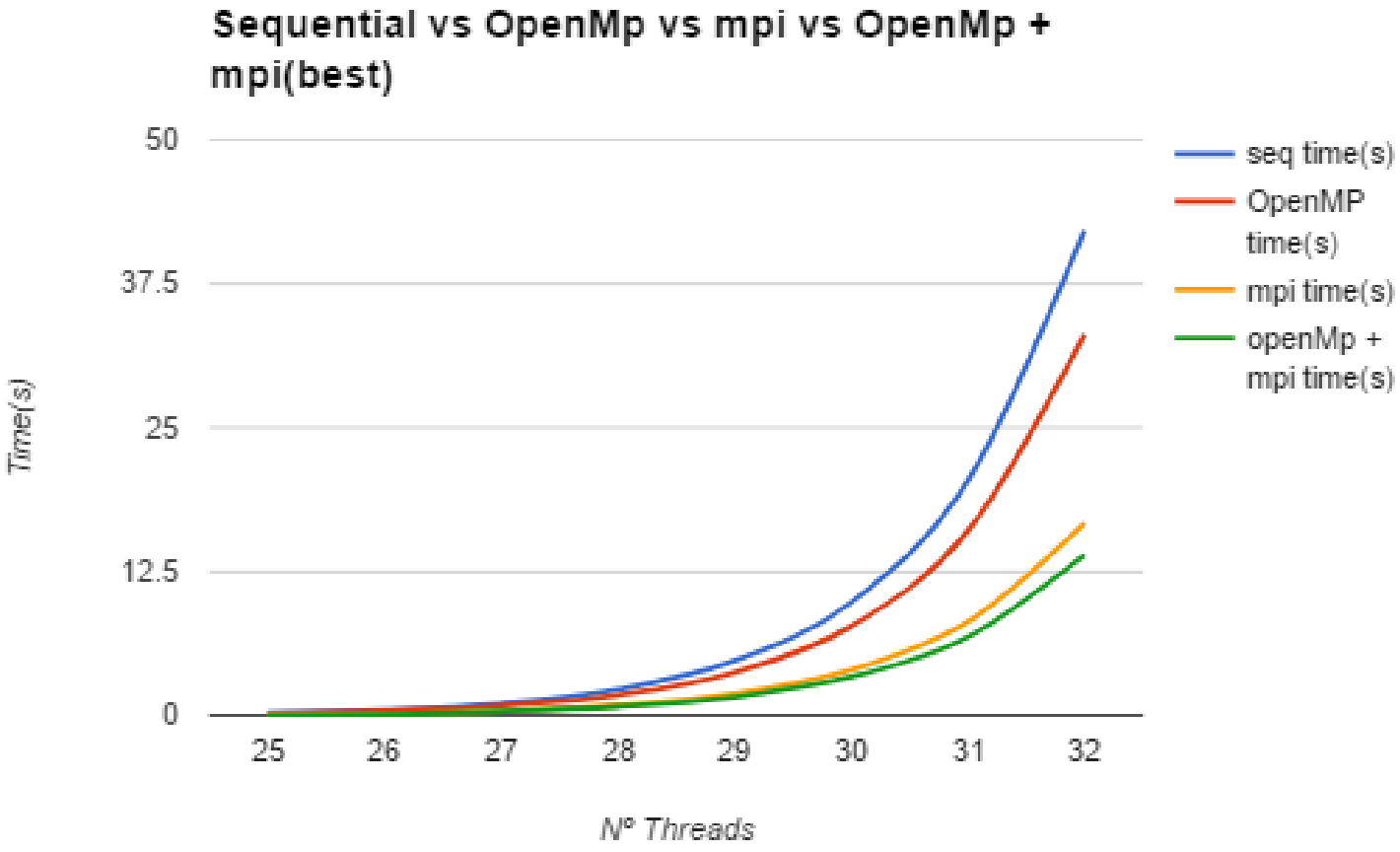

Figura 9: Comparação dos melhores tempos de cada versão.

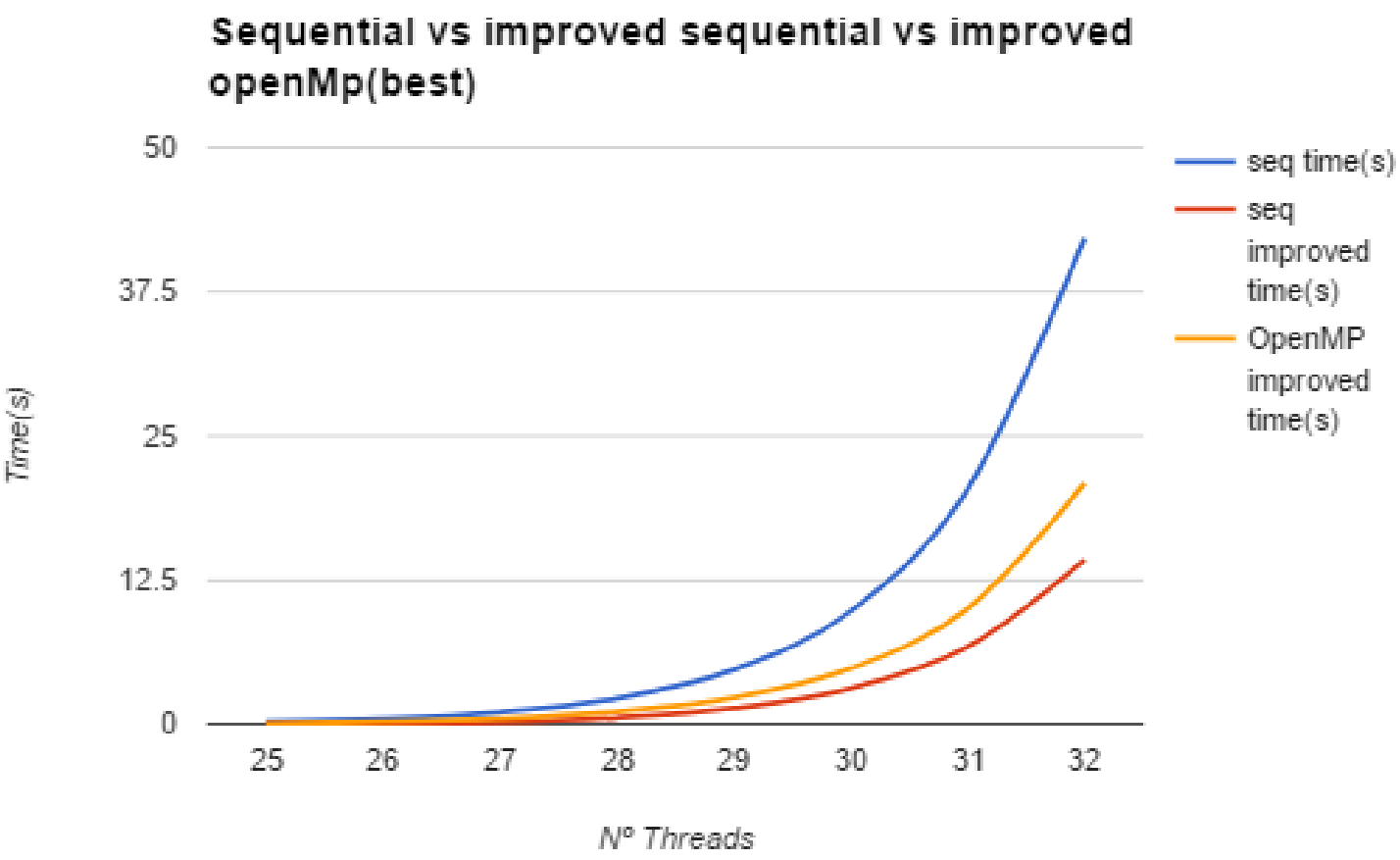

Figura 10: Comparação dos algoritmos melhorados com o sequencial base