

Instituto de Matemática e Estatística

EP1 - MAC0219

Cálculo do Conjunto de Mandelbrot em Paralelo com Pthreads e OpenMP

Professor: Alfredo Goldman

Alunos: Bruno Sesso

Gustavo Estrela de Matos

Lucas Sung Jun Hong

São Paulo, 2 de Maio de 2017

Conteúdo

1	Introdução	2
2	Código Sequencial	3
3	Código em Pthreads	7
3.1	Implementação com Divisão Estática	7
3.2	Implementação com Divisão Dinâmica	8
4	Código em OpenMP	12
4.1	Implementação com 1 for	17
4.2	Dynamic vs Static	22
5	OpenMP vs Pthreads	28
6	Discussão dos Resultados	31
7	Conclusão	36

1 Introdução

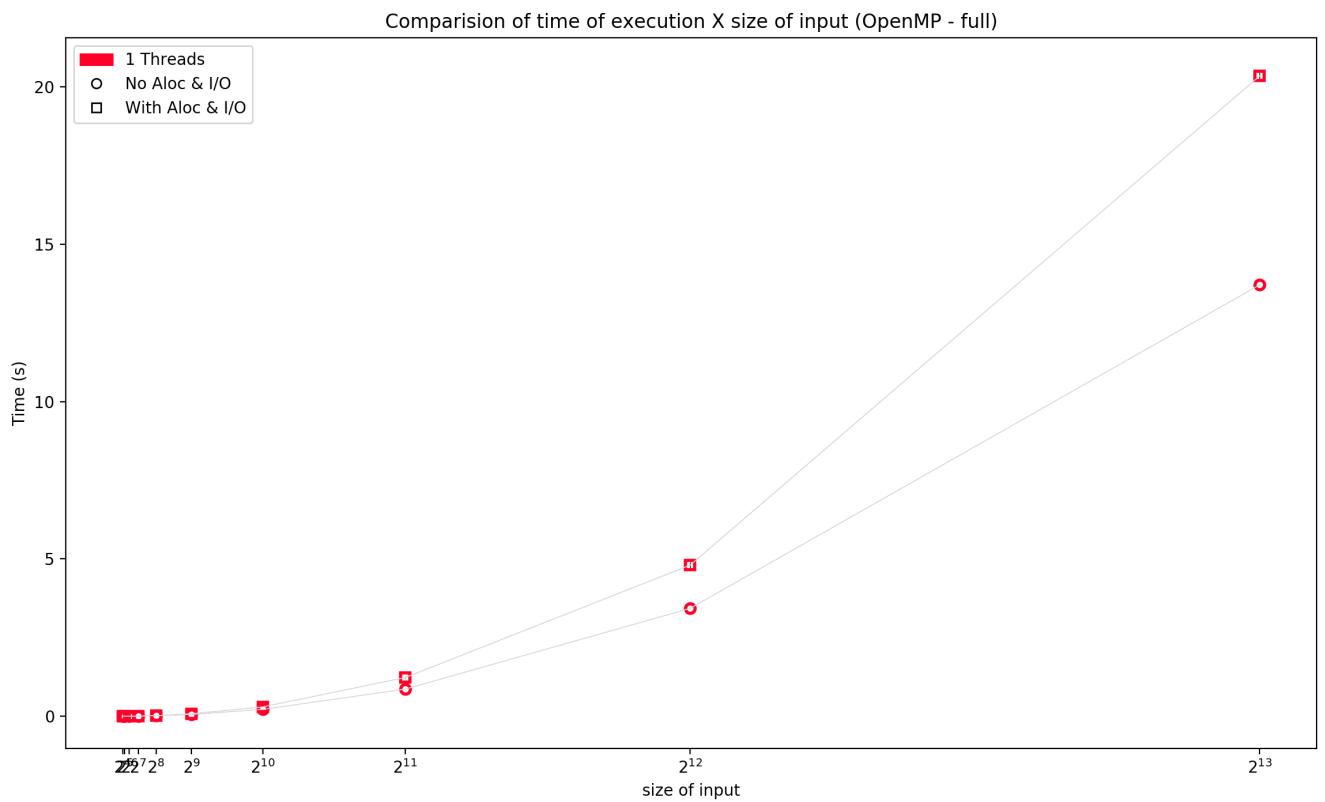
Esse trabalho tem como objetivo implementar e analisar duas versões paralelas de um código sequencial que é capaz de calcular o conjunto de mandelbrot para diversas regiões do plano complexo. As duas versões do código foram implementadas utilizando as bibliotecas OpenMP e PThreads.

Ao longo desse trabalho, iremos apresentar resultados de tempo de execução das diferentes implementações e suas respectivas variações. Para isso, utilizamos a ferramenta *perf*, capaz de realizar repetições de experimentos, apresentando resultados médios e com desvio padrão. Todos os resultados apresentados aqui foram feitos a partir de no mínimo 10 execuções do mesmo comando.

2 Código Sequencial

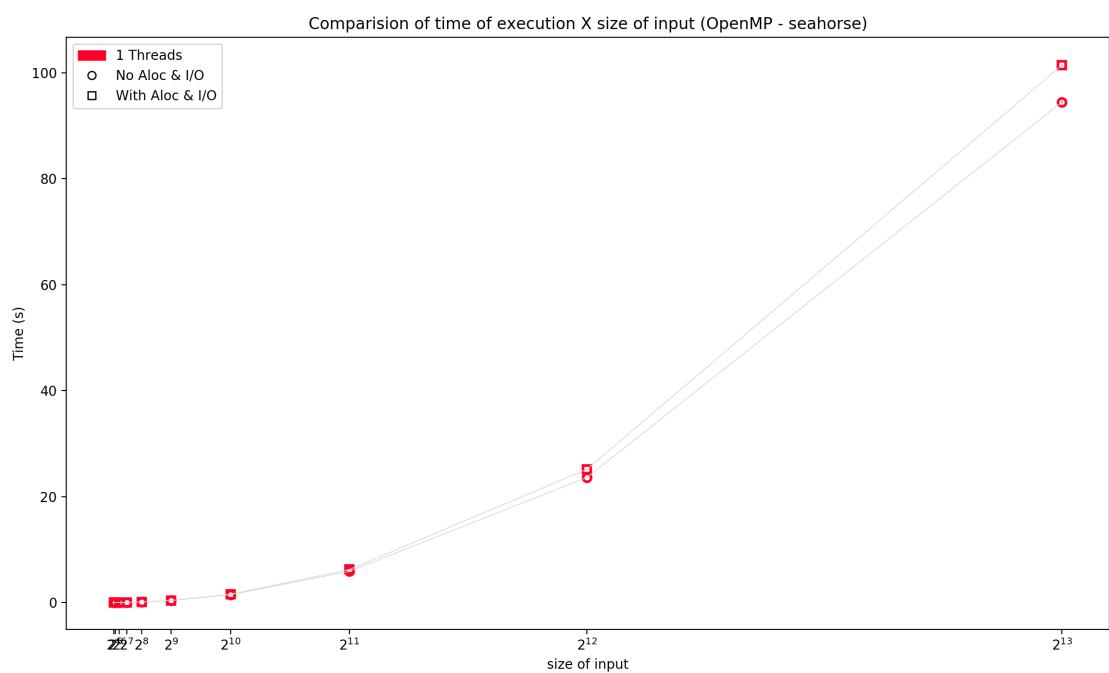
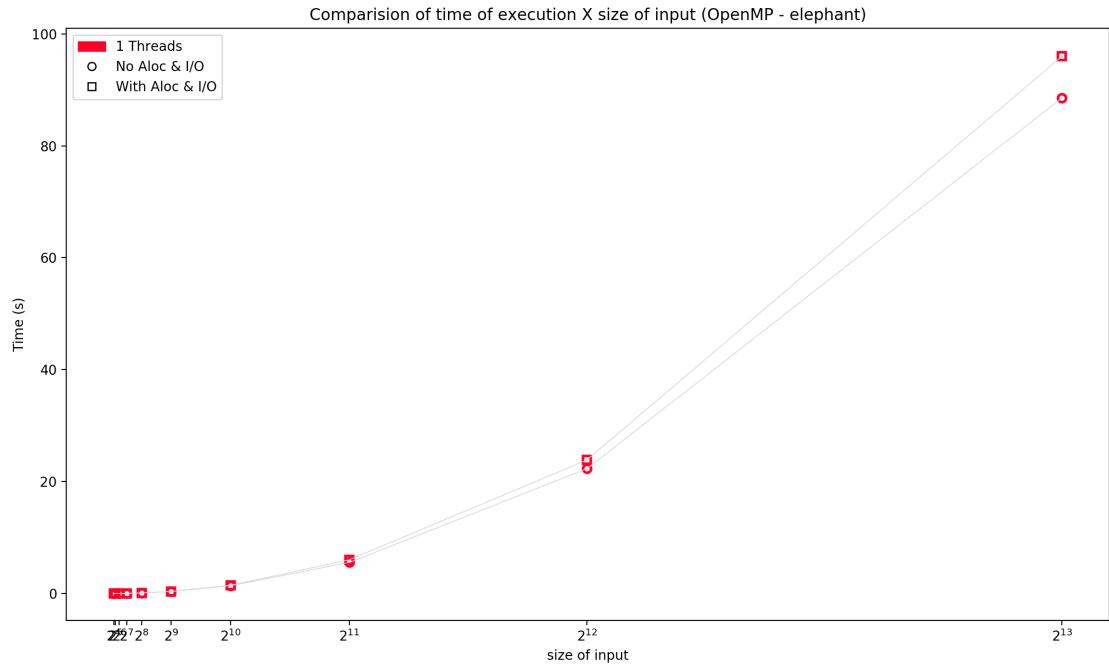
Para implementar a versão sequencial do programa do cálculo do conjunto de Mandelbrot temos uma versão com alocação de memória e com comandos de leitura e escrita e uma versão sem ambos. No caso sem ambos, como não há comandos de leitura, os comandos de leitura e escrita são retirados, removendo-se a função `write_to_file()`. Para a remoção de alocação de memória pode ser que o tempo de acesso a um vetor possa ser relevante, portanto não simplesmente removemos a função de alocação de memória. Para retirar as alocações, supomos que o tamanho máximo de uma imagem será de 11500px x 11500px e assim substituímos a alocação dinâmica do `image_buffer` por uma alocação estática: `char image_buffer[11500*11500]` [3].

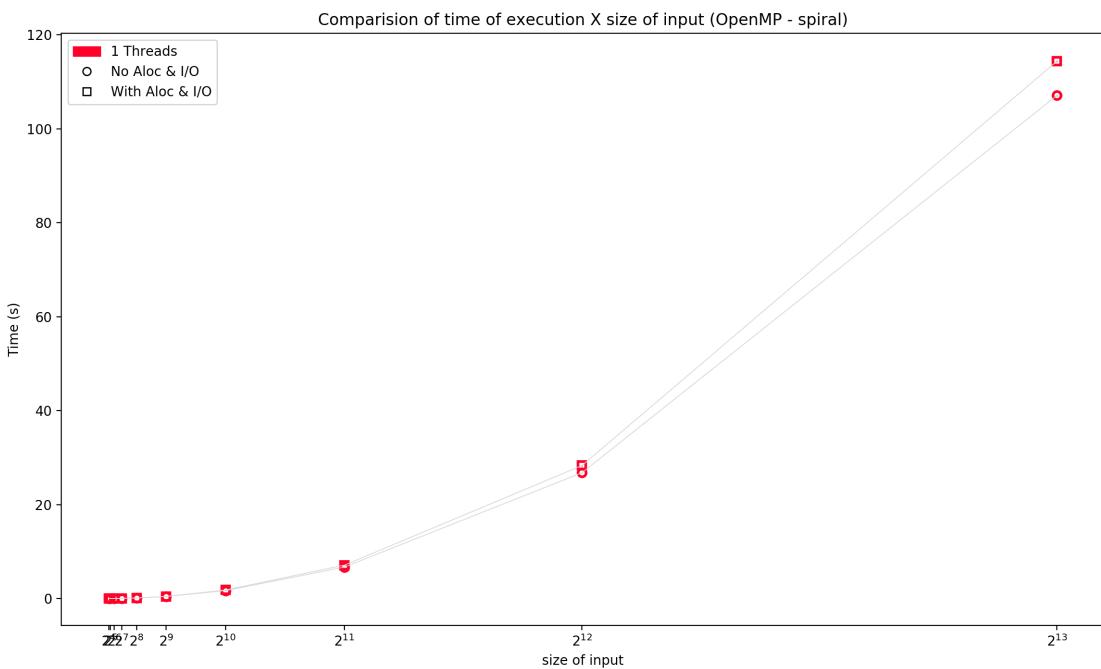
Comparemos os resultados obtidos nos testes com e sem alocação de memória e comandos de leitura e escrita:



Como esperado o tempo do programa sem alocação e sem comandos de leitura e escrita é

consideravelmente menor que a versão com. Nas demais regiões o comportamento se repete:





Nota-se que o tempo em cada região é diferente, pois em cada região pode haver mais cálculos de pontos com mais interações. Notemos ainda que a diferença de tempo para cada tamanho de entrada se mantém mesmo mudando-se as regiões. Por exemplo, a versão sem alocação dinâmica e sem operações de leitura e escrita executa cerca de 10 segundos mais rápido que a versão com, dado a entrada de tamanho 2^{13} , independente da região. Isso ocorre, pois comandos de leitura e escrita e alocação dinâmica, dependem do tamanho da entrada e independem do número de interações do cálculo para determinado ponto.

Para cada implementação também vemos que o tempo para calcular a região Full foi bastante menor que para as demais regiões:

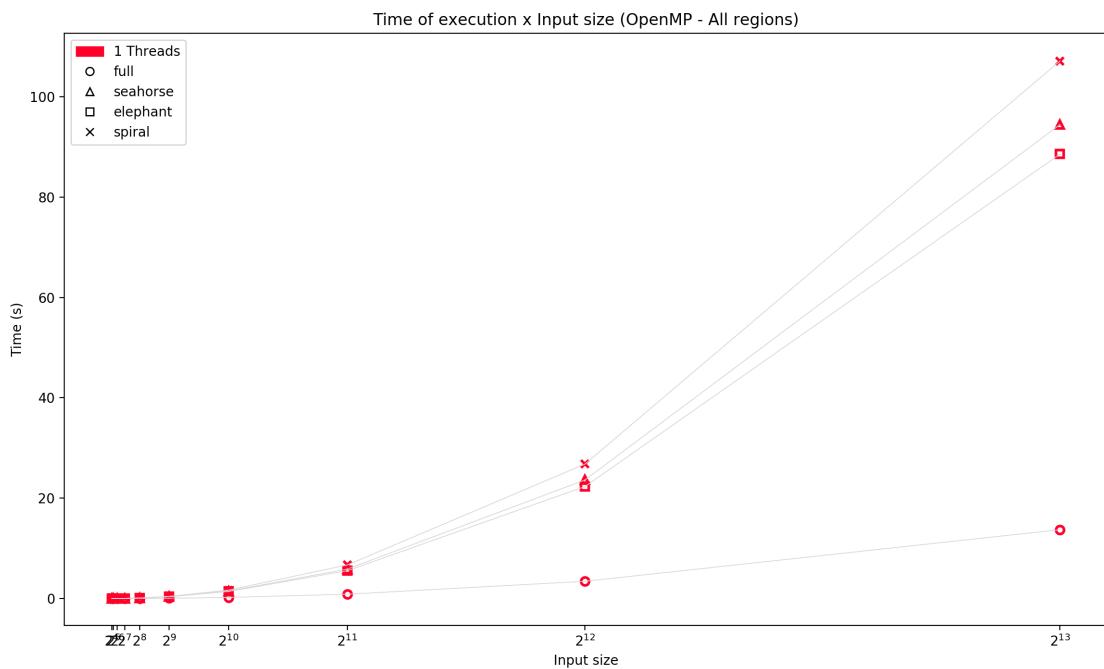


Figura 1: Implementação sem Operações de leitura e escrita e sem alocação de memória

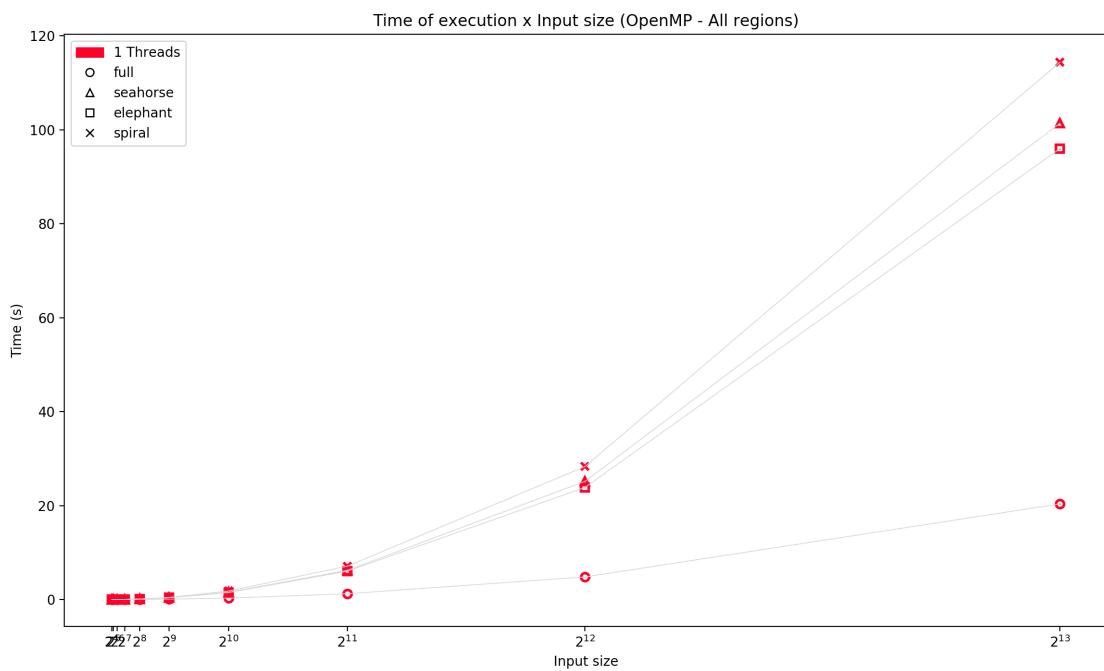


Figura 2: Implementação com Operações de leitura e escrita e com alocação de memória

3 Código em Pthreads

Utilizamos duas diferentes abordagens para paralelizar o código com o uso de pthreads, tentando se aproximar ao comportamento do OpenMP e suas diretivas *omp parallel for schedule (dynamic)* e *omp parallel for schedule (static)*. As diferenças de funcionamento dos dois códigos está na maneira em que o trabalho é dividido e como ele é distribuído para cada thread.

3.1 Implementação com Divisão Estática

Chamamos de implementação com divisão estática a versão do nosso código em pthreads no qual o trabalho é dividido em n pedaços de mesmo tamanho, e cada pedaço é dado a uma thread. Chamamos essa implementação de estática porque cada thread recebe apenas um bloco de trabalho, pré-determinado pela divisão feita, que será processado do começo ao fim (no escopo da thread) pela mesma thread.

Para implementar esse código, precisamos apenas construir uma estrutura de dados que era capaz de guardar um bloco de pixels a ser calculado. Dado essa estrutura, basta criar uma thread para cada bloco de trabalho, que por sua vez deve calcular os pixels correspondentes e atualizar o buffer de cores.

Devemos observar que essa implementação pode implicar em threads ociosas enquanto outras estão trabalhando. Como a quantidade de iterações necessárias para se calcular o valor de um pixel varia, é possível que um bloco de pixels seja calculado muito mais rápido do que outro; imagine por exemplo um bloco onde cada ponto calculado diverge rapidamente e outro bloco onde isso não acontece. Portanto, como a divisão de trabalho é estática, é provável que uma thread termine seu trabalho muito antes de outra, o que significa em um uso não muito bom de recursos da máquina.

Uma possível solução para esse problema seria o aumento no número de threads, o que cria uma fragmentação maior do trabalho. Essa fragmentação ameniza o problema anterior porque divide mais o trabalho, deixando menor a diferença de tempo necessário para se calcular cada parte. Entretanto, criar um número excessivo de threads pode dar mais trabalho ao

escalonador do sistema operacional, que deve lidar com várias linhas de processamento.

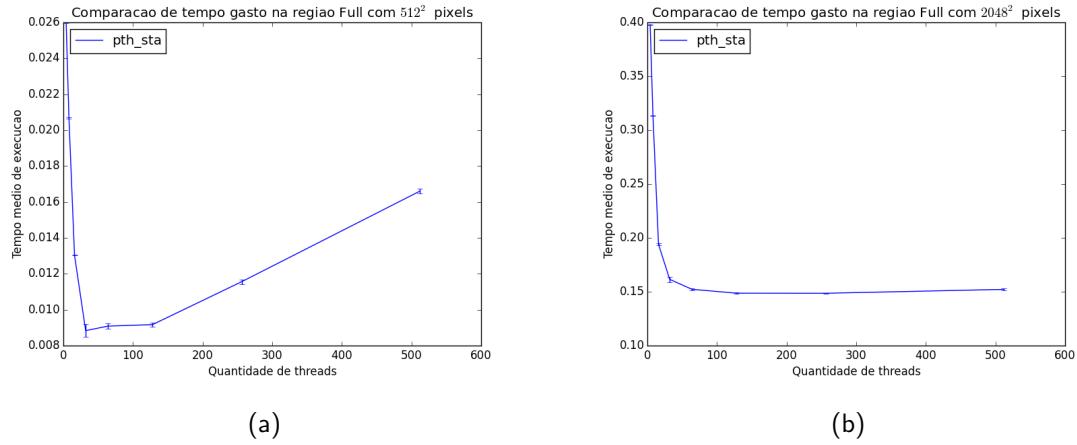


Figura 3: É possível notar em ambas figuras que o aumento do número de threads de fato diminui o tempo de execução do programa. No gráfico 3(a) fica evidente que o tempo gasto no controle das threads pode afetar o tempo de execução do programa.

3.2 Implementação com Divisão Dinâmica

A implementação dinâmica do nosso programa também divide o trabalho em n pedaços, entretanto agora n não é mais, necessariamente o número de threads disponíveis. Após dividir o trabalho em pedaços, o nosso programa agora é capaz de, dinamicamente, delegar blocos de pixels a cada thread. Portanto, agora diminuímos o problema de threads ociosas, porque podemos dividir mais os blocos de trabalho e sempre que uma thread termina um bloco, podemos dar a ela um novo bloco para computar (desde que ainda haja trabalho a ser feito).

Veja abaixo um pseudo-código para esse programa:

```

1: function COMPUTEMANDELBROT
2:    $S \leftarrow$  lista de nacos de todos os pixels
3:   while  $S \neq \emptyset$  do
4:      $chunk \leftarrow S.popChunk ()$ 
5:     Espere alguma thread estar livre
6:     for all Thread  $T$  do
7:       if  $T$  está livre then
8:          $T.compute (chunk)$ 

```

```

9:         end if
10:        end for
11:    end while
12: end function

```

A implementação desse código é um pouco mais complicada, porque depende, além da estrutura de dados já usada na implementação estática, de um maior controle de concorrência. O conceito principal utilizado é o de *condition variable*, que nos permite colocar a linha principal de execução em espera, enquanto as threads calculam os pixels, para voltar a ser executada quando alguma thread estiver livre.

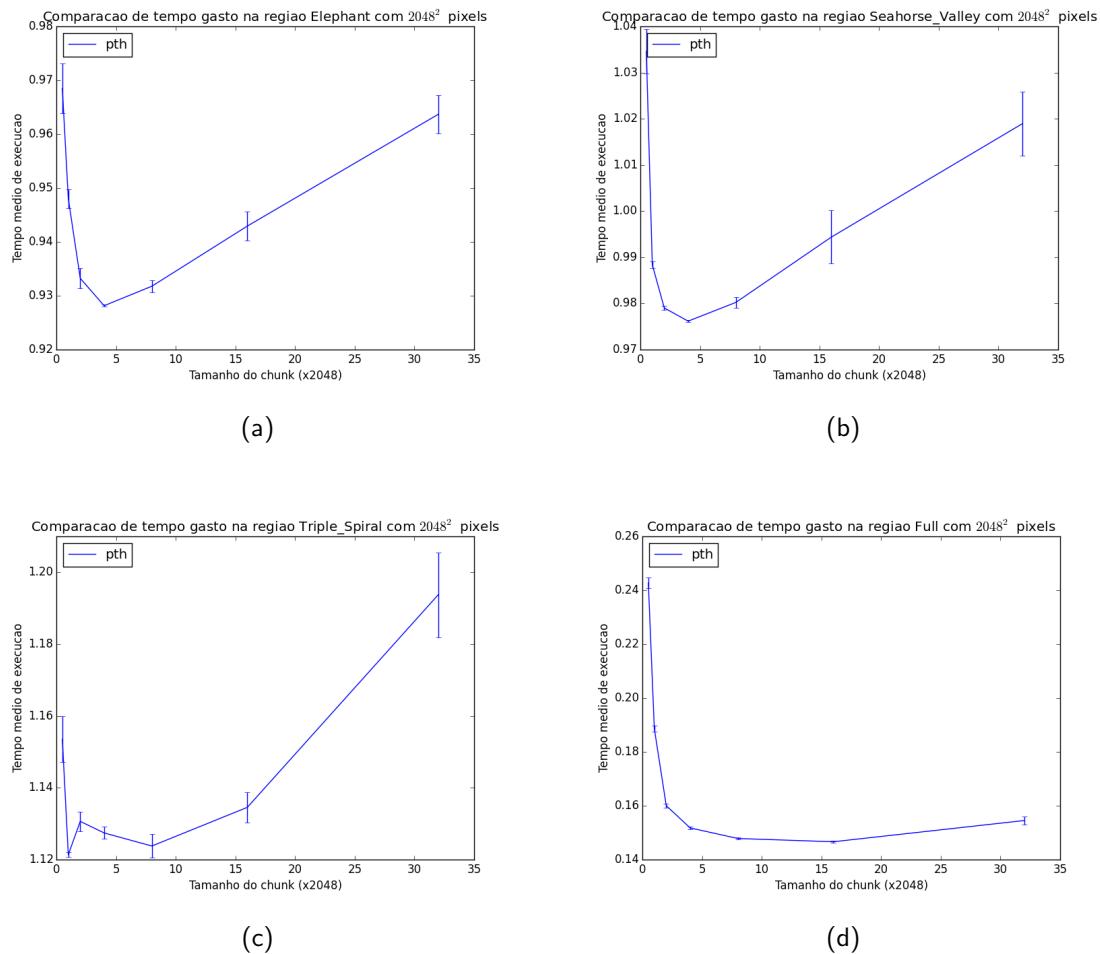


Figura 4: Verificamos empiricamente que definir o tamanho do chunk como o tamanho do lado da imagem traz bons resultados para o tempo de execução.

Diminuído o problema de threads ociosas, é esperado que essa implementação seja mais rápida do que a anterior, porém, é necessário notar que o nosso código ficou mais complexo,

e exige mais recursos computacionais para o controle das threads. Esse tempo pode ser prejudicial se o tamanho do chunk for pequeno, pois nessa situação há muitas trocas de chunks em cada thread, fazendo com que a maior parte do tempo de processamento seja gasto no controle das threads. Por outro lado, se temos chunks muito grandes, o tempo gasto deve aumentar, porque caímos novamente no problema da implementação static, onde o trabalho não era bem dividido. Depois de fazer alguns testes, verificamos que definir o tamanho do chunk igual a quantidade de pixels em uma linha da imagem nos dava bons resultados; inclusive, esse foi o mesmo valor adotado em nossa implementação com OpenMP.

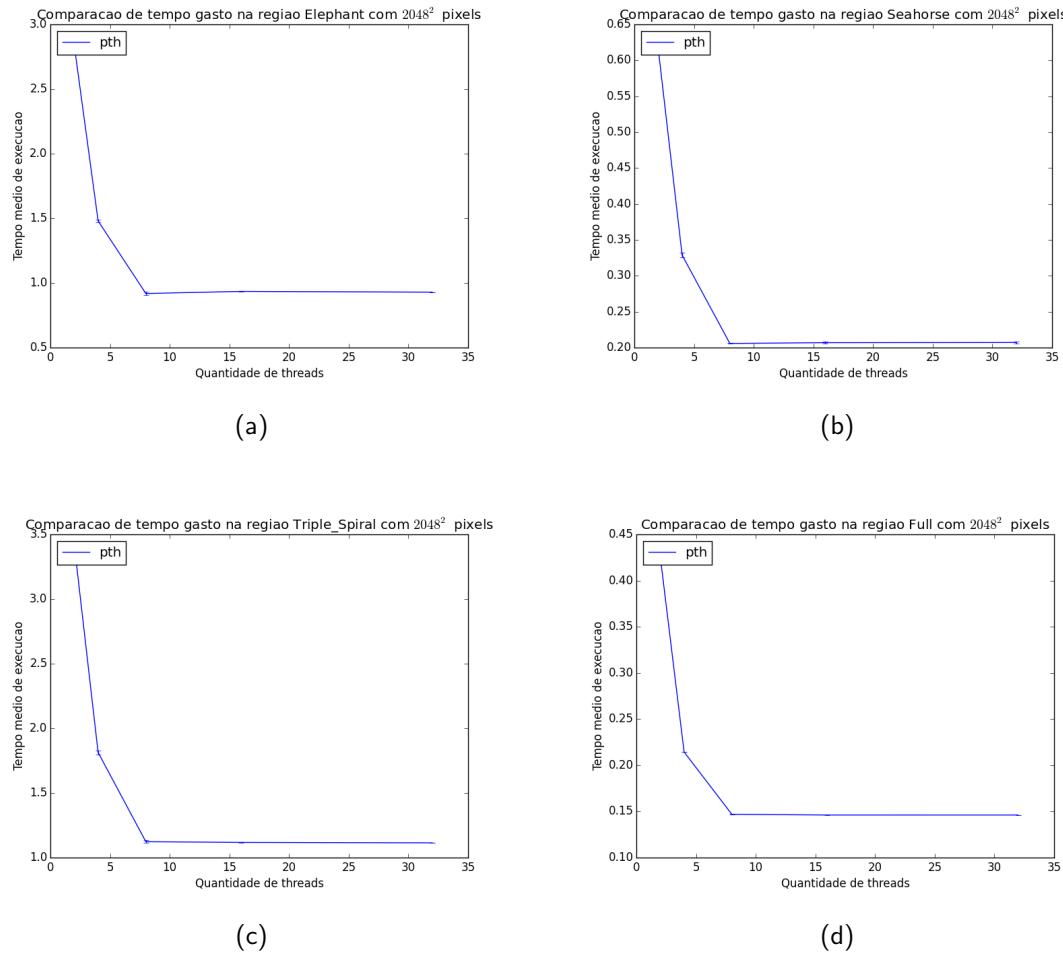


Figura 5: Podemos observar que existe uma saturação de melhora de desempenho quando o número de threads atinge o número de cores (8) da máquina.

Como distribuímos dinamicamente os blocos de trabalho, é esperado que o aumento do número de threads não melhore o problema de threads ociosas, diferente do que vimos na implementação com divisão estática e na figura 3. A melhora com o aumento de threads deve

vir unicamente do melhor uso da quantidade de cores da máquina, ou seja, a quantidade de threads deve melhorar o desempenho enquanto não foi maior do que o número de processadores da máquina. De fato, observamos esse comportamento na figura 5

Veja abaixo a comparação em consumo de tempo entre as duas implementações de paralelismo com Pthreads:

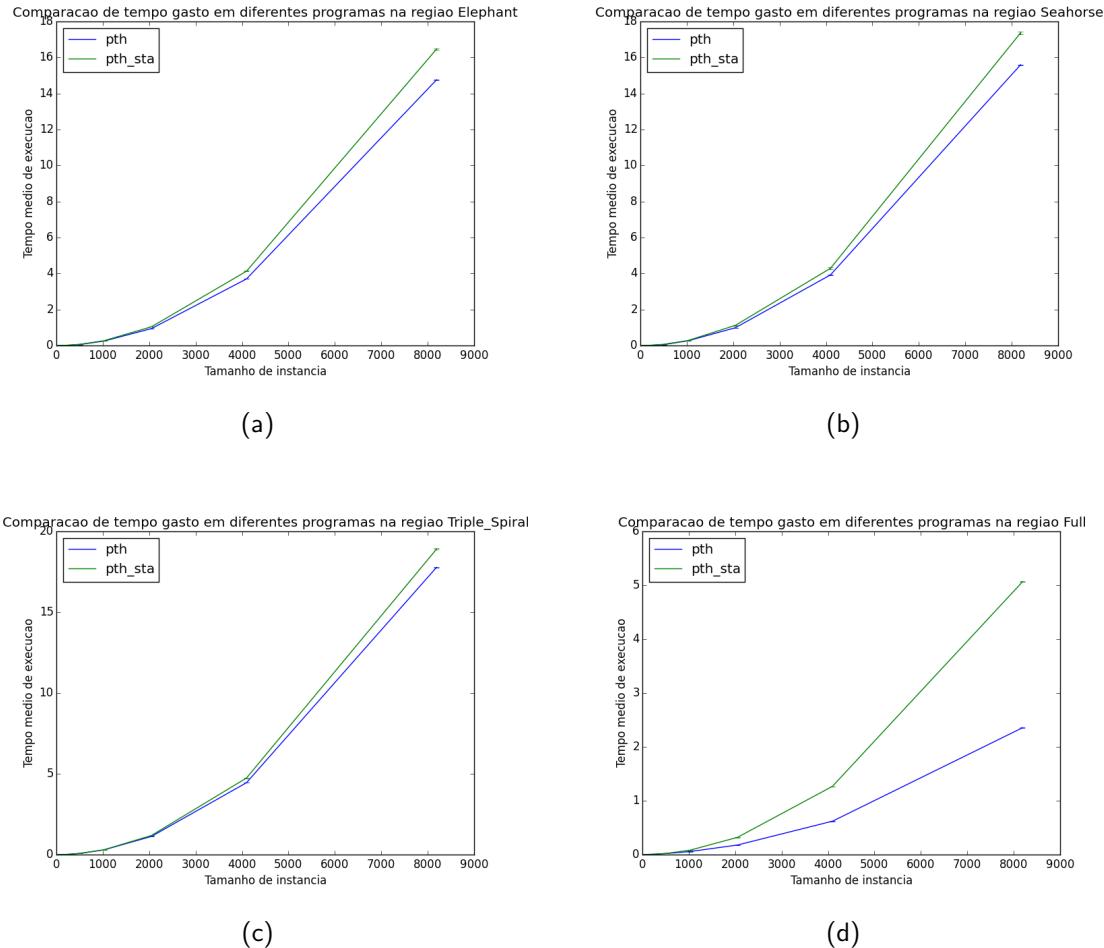


Figura 6: Podemos observar nessas figuras que a implementação com divisão dinâmica possui desempenho melhor do que estático e que, além disso, essa diferença tende a aumentar quando o tamanho da instância também aumenta. É provável que as regiões onde o trabalho está. Esses testes foram realizados com número de threads igual a quantidade de processadores da máquina.

4 Código em OpenMP

Para implementar a versão OpenMP do programa do cálculo do conjunto de Mandelbrot tivemos que remover a alocação de memória e comandos de leitura e escrita do código fornecido previamente. Como não há comandos de leitura, os comandos de leitura e escrita são retirados, removendo-se a função `write_to_file()`. No caso de alocação de memória pode ser que o tempo de acesso a um vetor possa ser relevante, portanto não simplesmente removemos a função de alocação de memória. Para retirar as alocações, supomos que o tamanho máximo de uma imagem será de 11500px x 11500px e assim substituímos a alocação dinâmica do `image_buffer` por uma alocação estática: `char image_buffer[11500*11500]` [3].

Para paralelizar o cálculo, como esperado do OpenMP, é tão fácil como adicionar um `#pragma` antes do `for` que realiza o cálculo para cada pixel da imagem:

```
1 #pragma omp parallel for private (...) num_threads(nThreads) schedule(  
2     dynamic)  
3  
4     for (i_y = 0; i_y < i_y_max; i_y++) {  
5         c_y = c_y_min + i_y * pixel_height;  
6         ...
```

Implementando dessa forma obtemos os seguintes resultados:

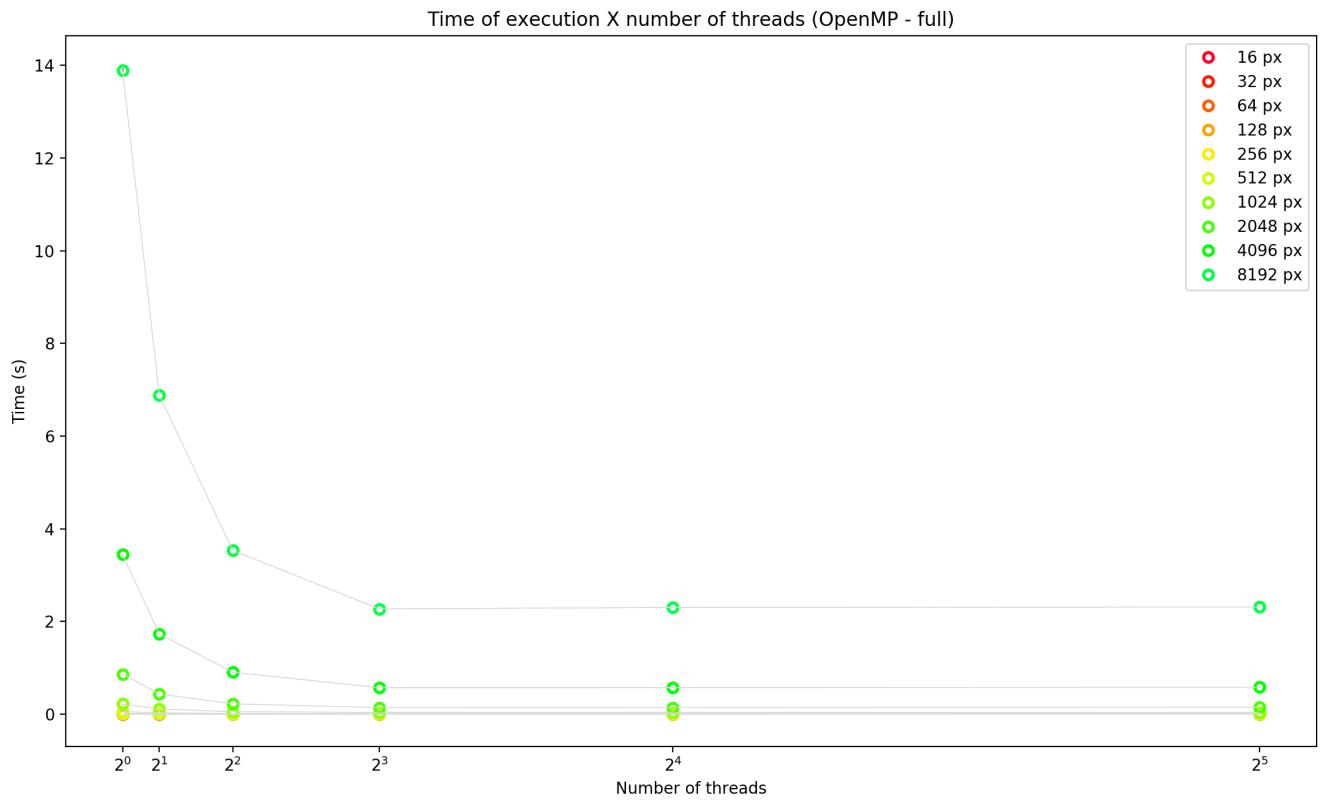
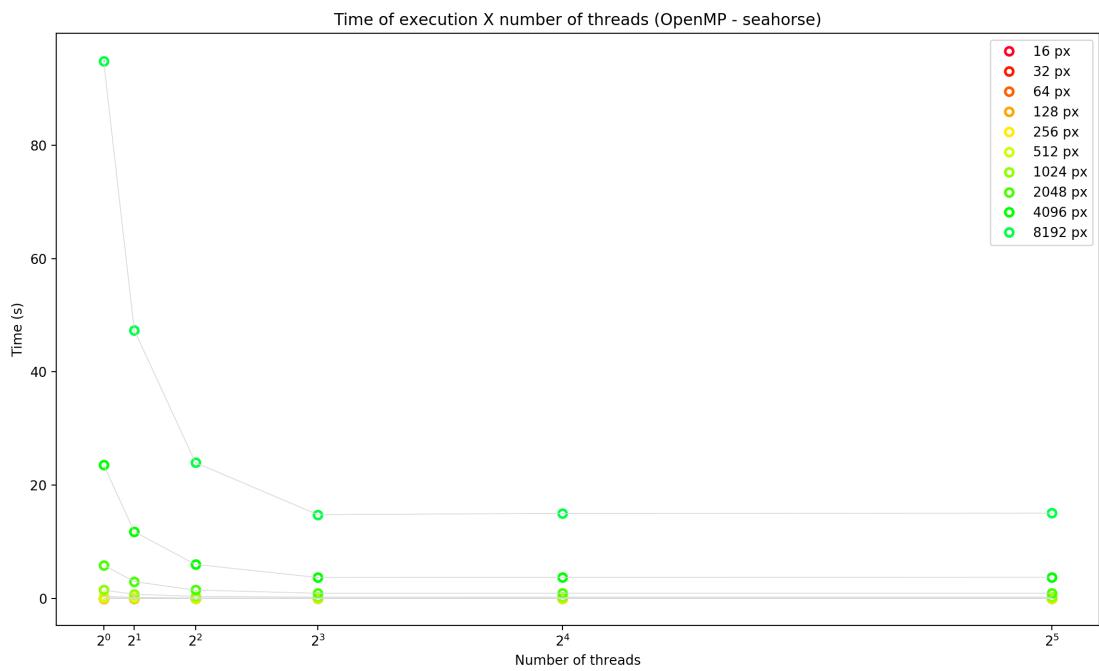
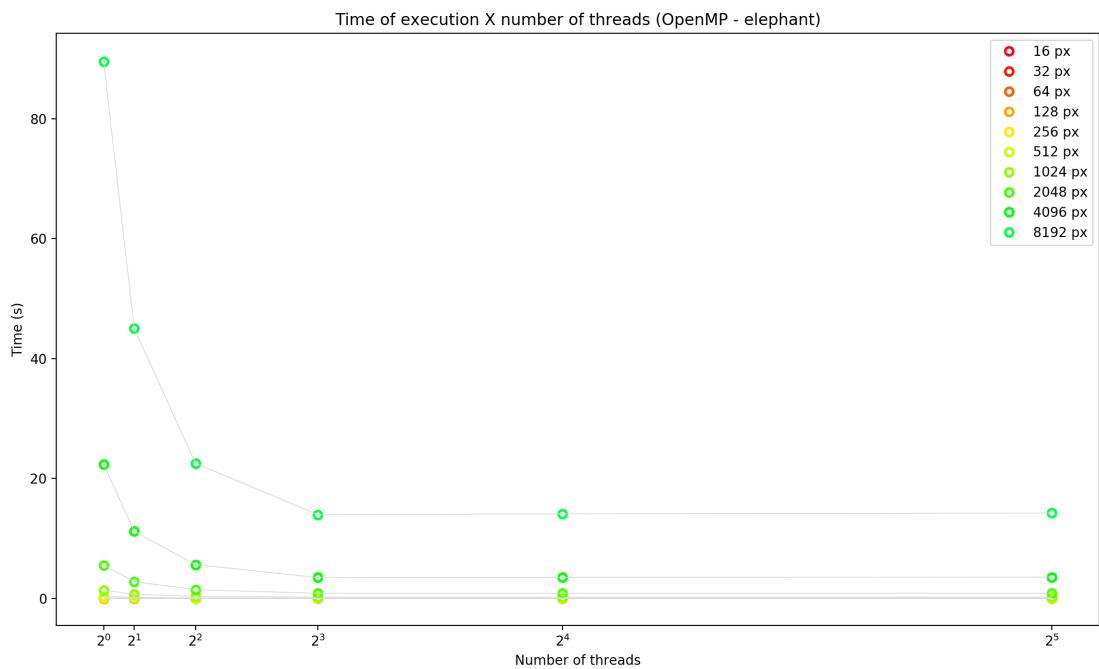
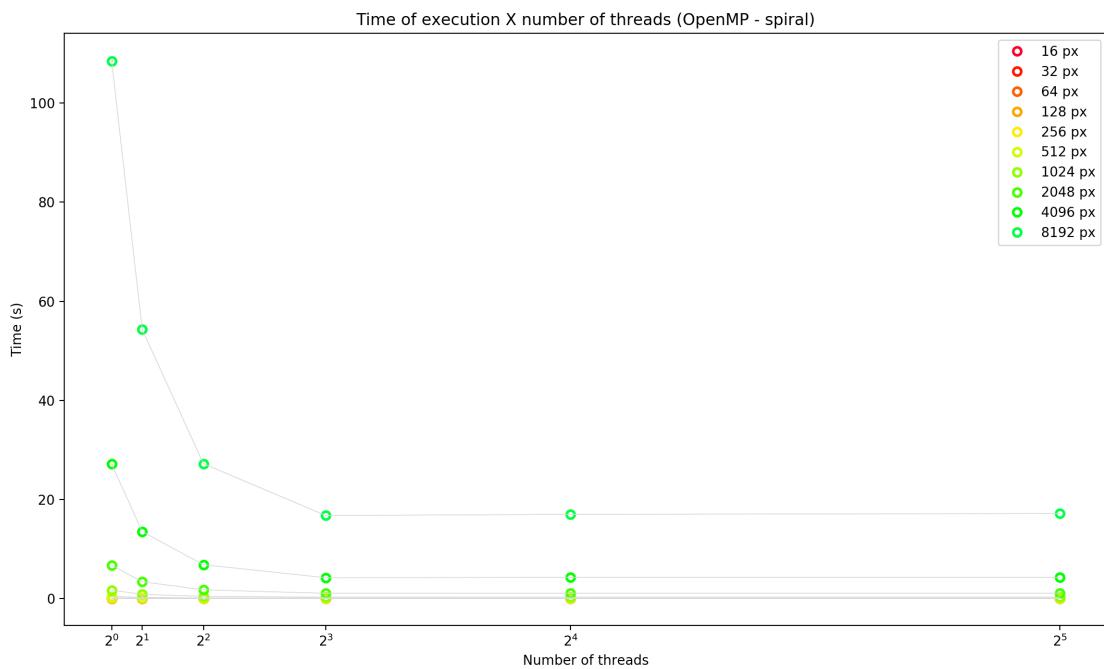


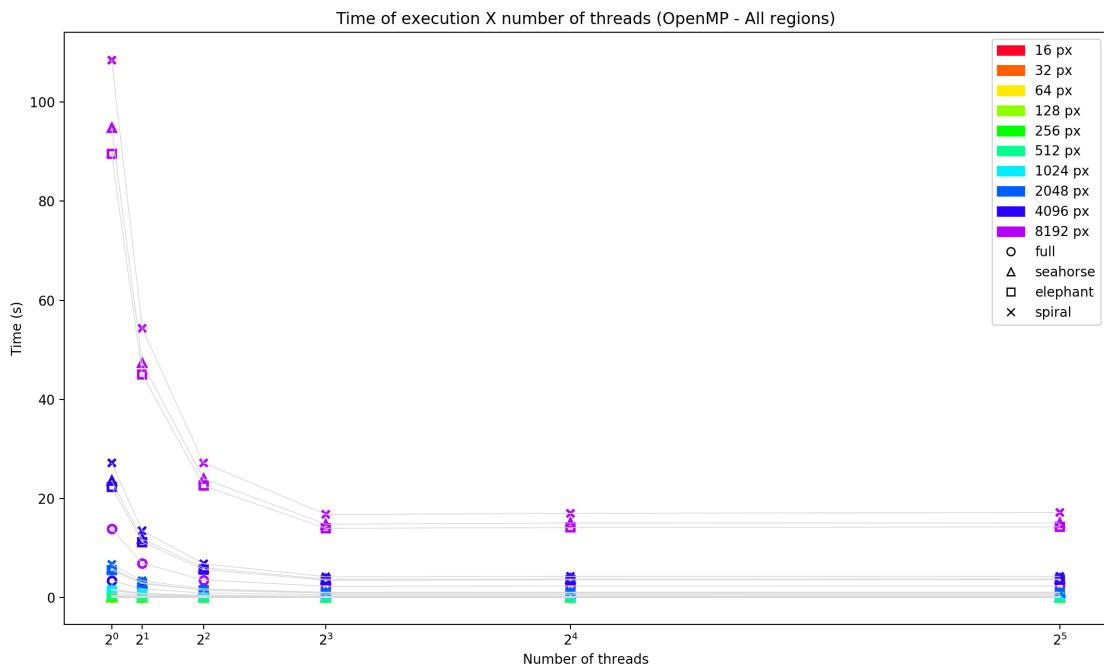
Figura 7: Esse gráfico mostra o tempo de execução X numero de threads para cada tamanho de entrada. Esses valores foram experimentados na região "full". Barras de erro também estão plotadas, porém são tão pequenas que talvez não estejam visíveis.

Percebemos que o aumento de threads de fato melhorou o tempo de execução do programa. No entanto ao ultrapassar 8, o mesmo número de cores da máquina testada, não houve grande melhora no tempo de execução. Esse mesmo comportamento foi observado para as demais regiões:



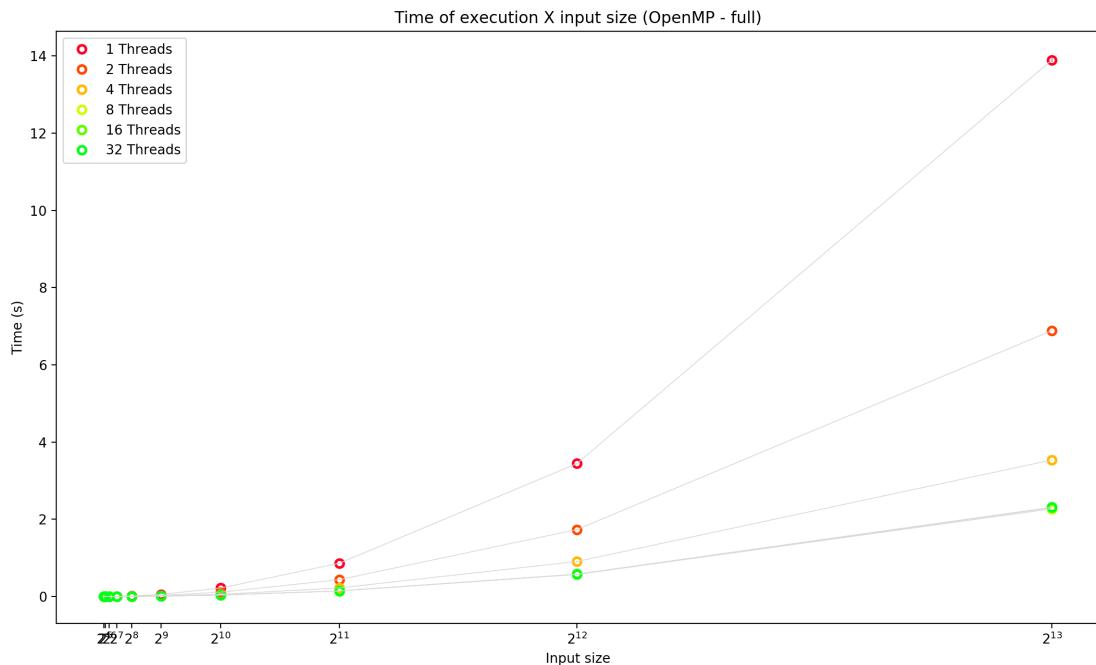


Nota-se que nessas regiões apesar do comportamento do tempo X numero de threads manter-se o mesmo, o tempo de execução aumenta, pois nessas regiões há mais cálculos de pontos com mais interações. Podemos comparar todas as regiões:



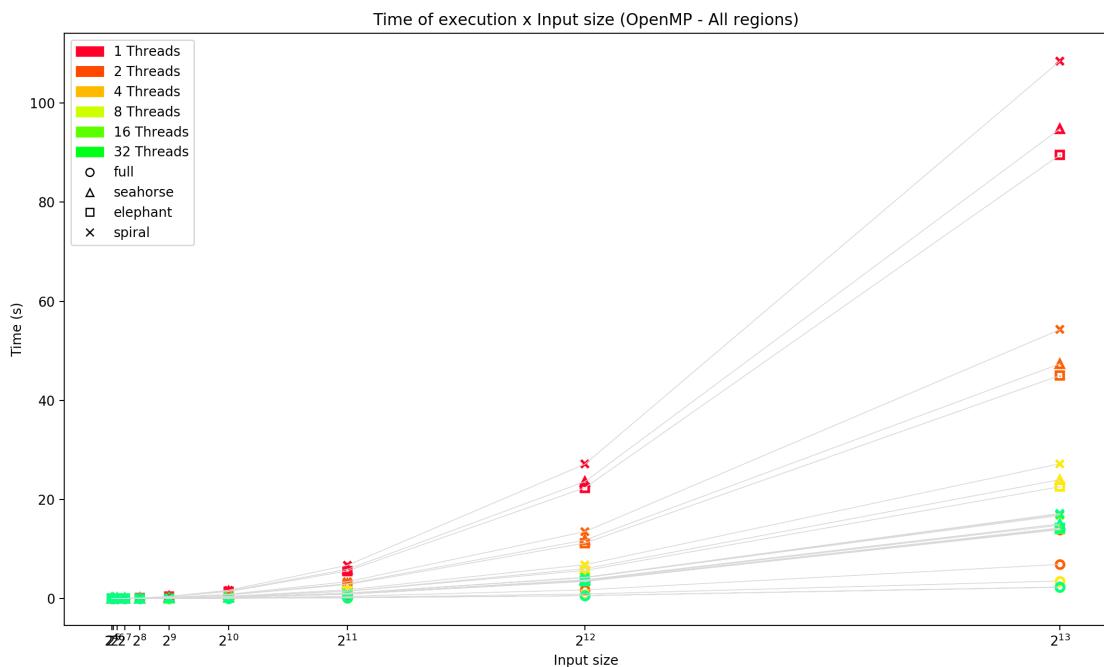
Vemos que as regiões Spiral, Seahorse e Elephant tem tempo de execução bastante maior que a região Full.

Em todos esses gráficos também vemos que quanto maior o tamanho da entrada, mais tempo o programa leva para ser executado. Porém também queremos saber qual é a relação entre tamanho de entrada e o tempo:



Vemos que há um aumento exponencial do tempo pelo tamanho da imagem. Vale lembrar que consideramos o tamanho de entrada, somente o tamanho do lado da imagem, portanto o comportamento exponencial se justifica por o número de pixels (chamadas para a função `compute_manderbolt()`) ser o tamanho do lado da imagem elevado ao quadrado.

Nas demais regiões o comportamento é semelhante:



4.1 Implementação com 1 for

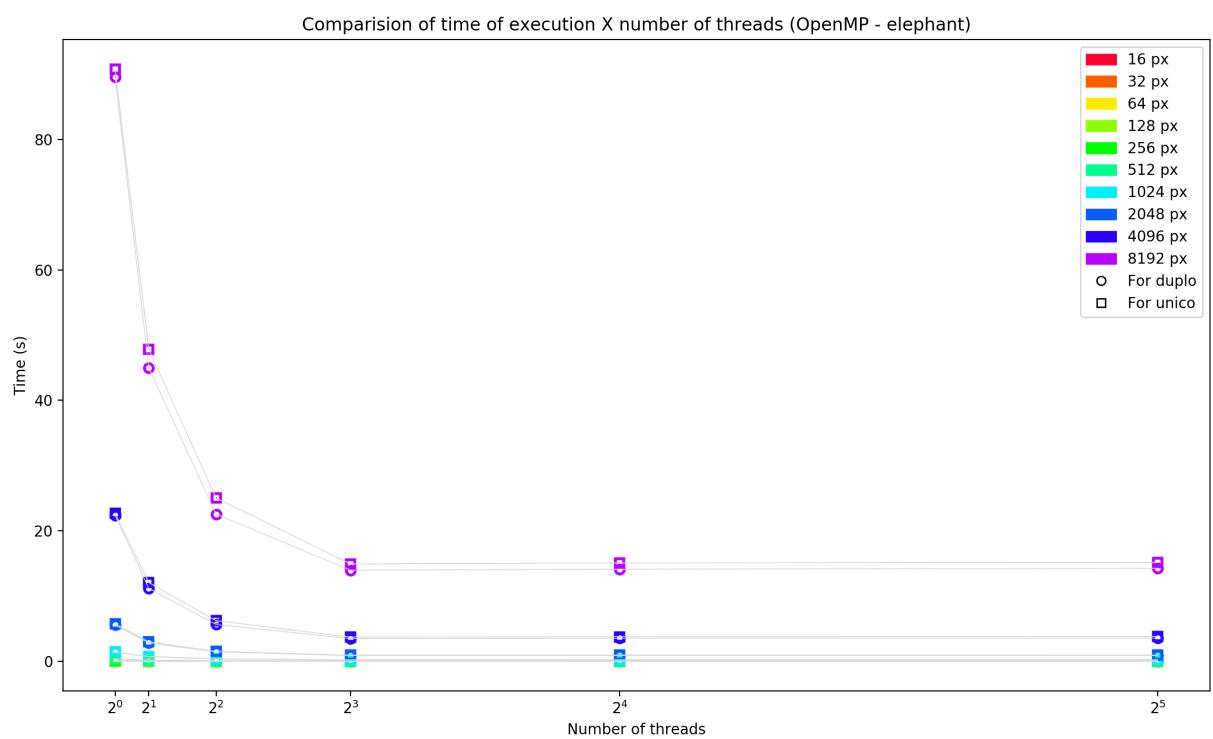
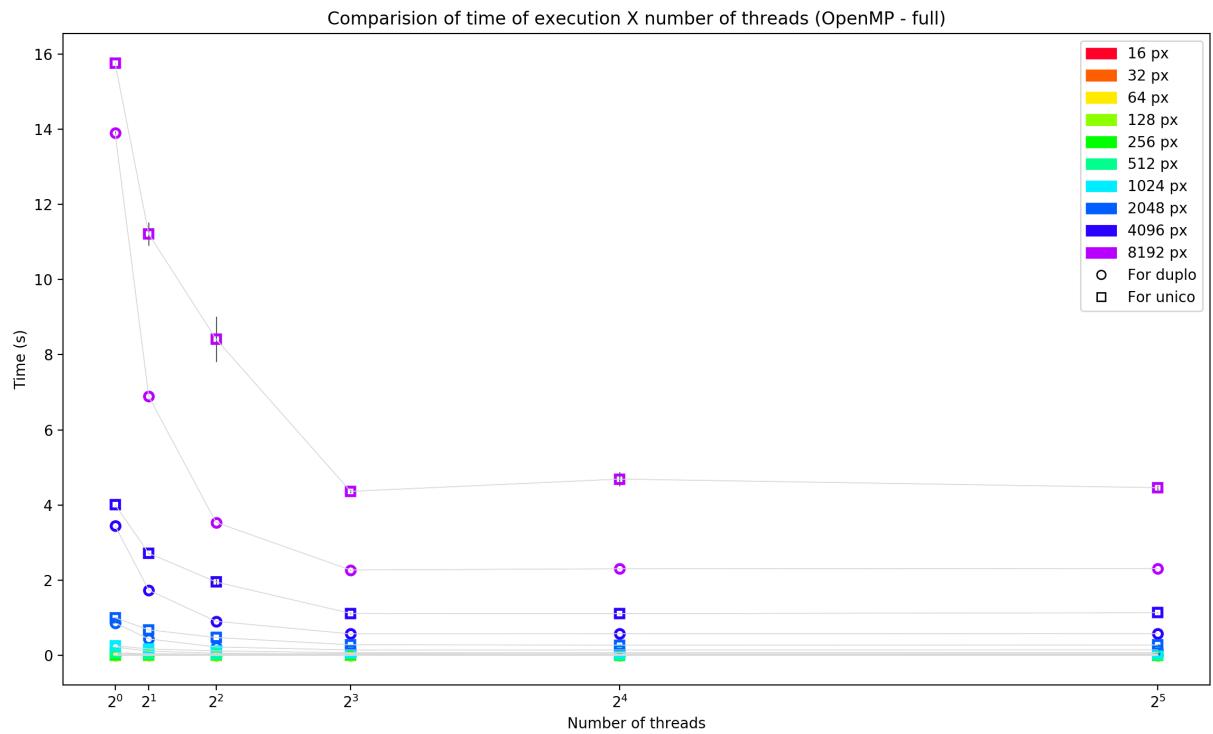
Após realizar esses primeiros testes imaginamos que se paralelizássemos o for de dentro o programa pudesse ficar mais rápido. Para fazer isso como eram dois fors simples, decidimos transforma-los em um só:

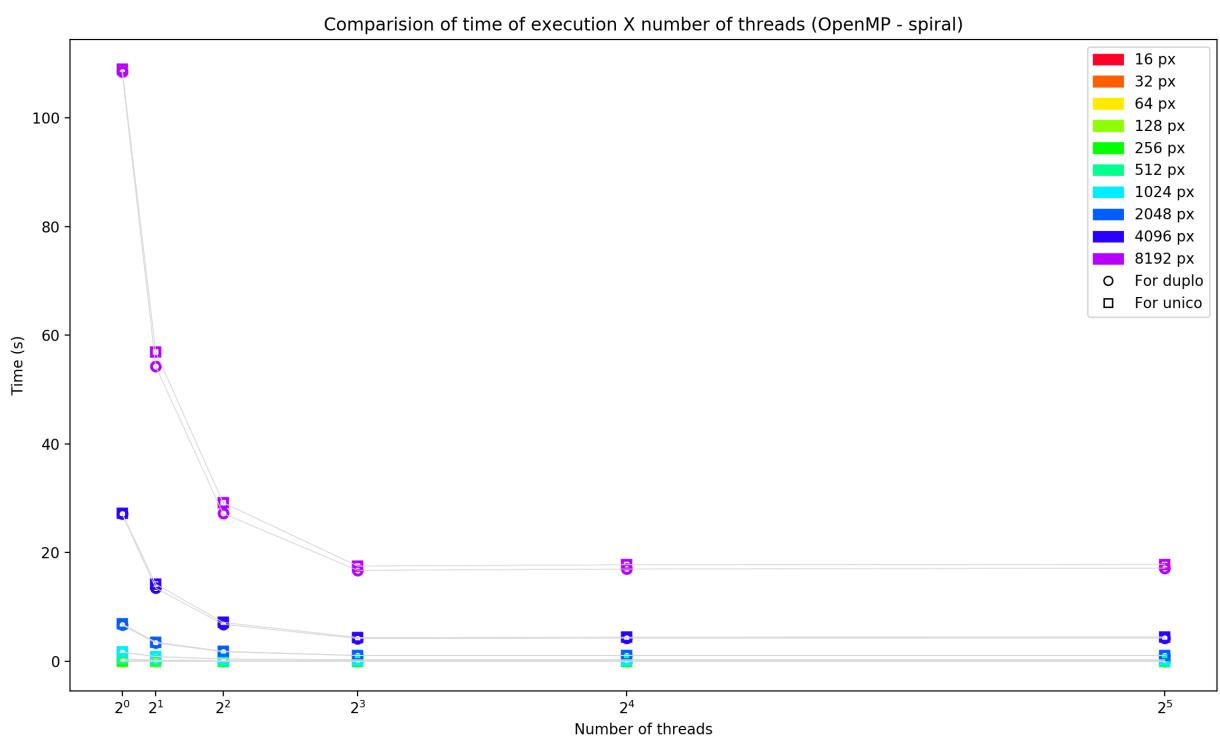
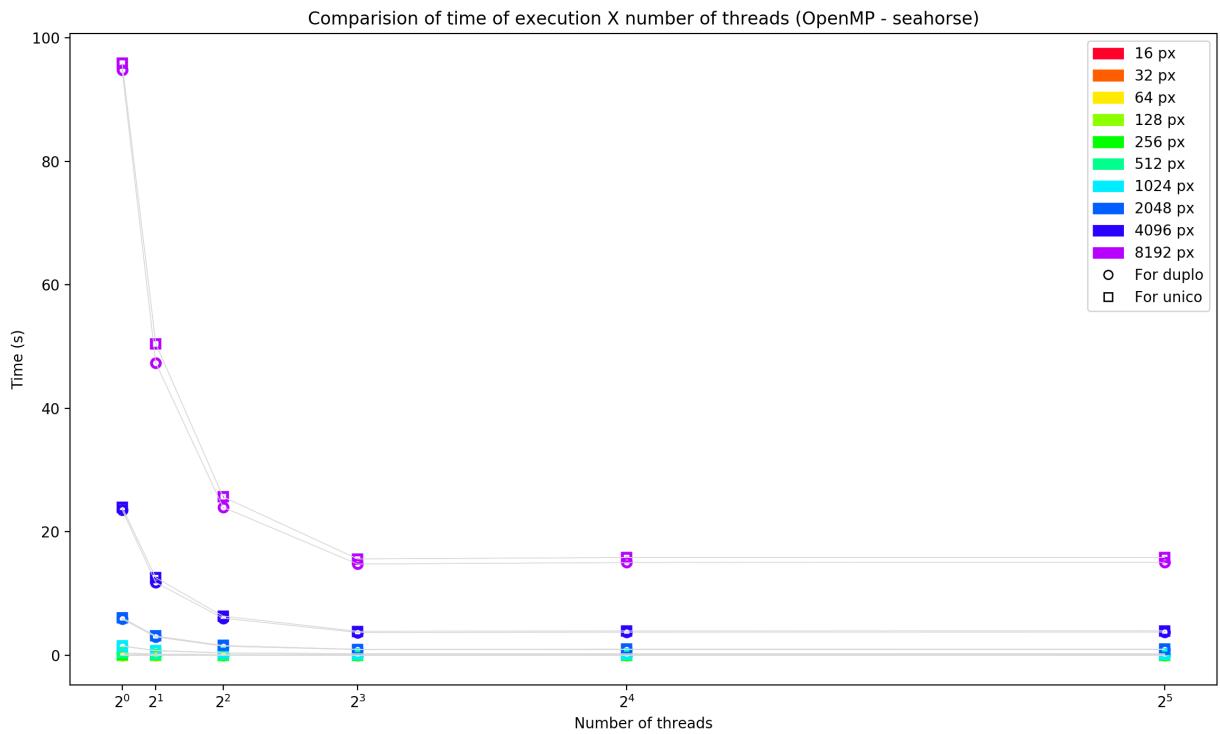
```

1 #pragma omp parallel for private(...) num_threads(nThreads) schedule(
2     dynamic)
3     for (i = 0; i < i_y_max * i_x_max; i++) {
4         i_y = i / i_y_max;
5         i_x = i % i_y_max;
6         ...

```

Com essa implementação e usando `schedule(dynamic)`, ou seja com o chunk de tamanho 1, obtivemos os seguintes resultados:



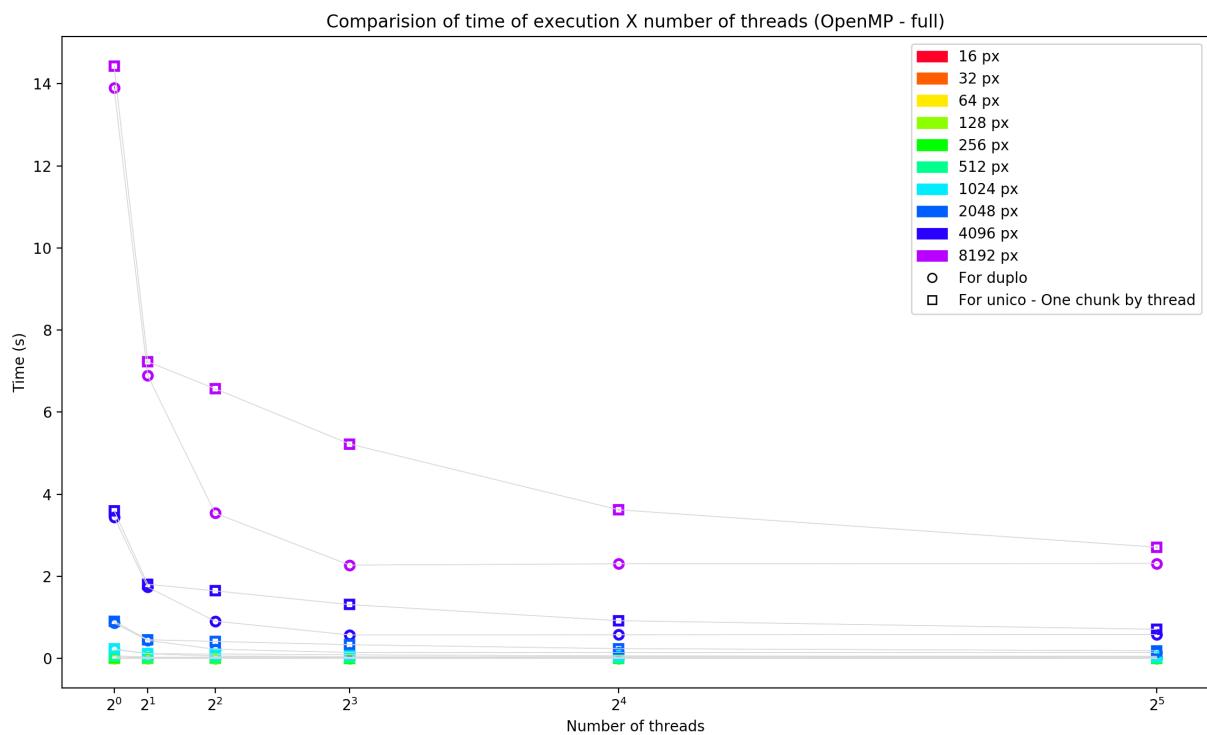


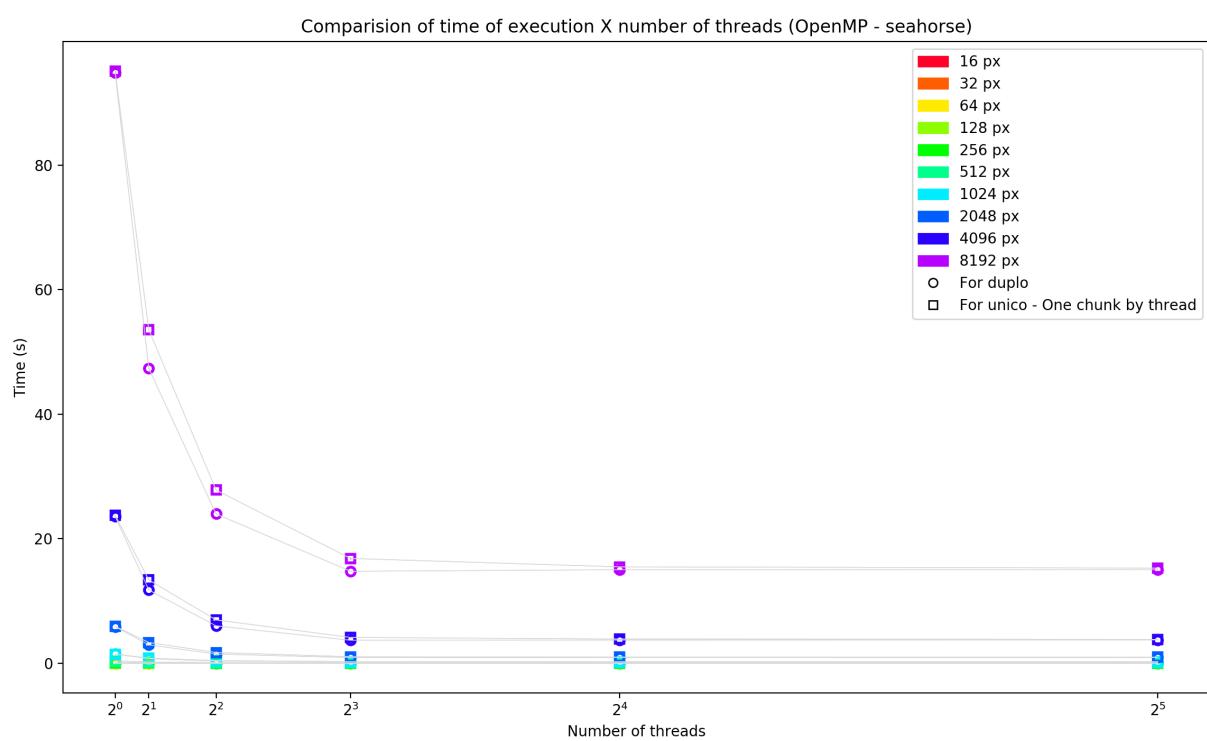
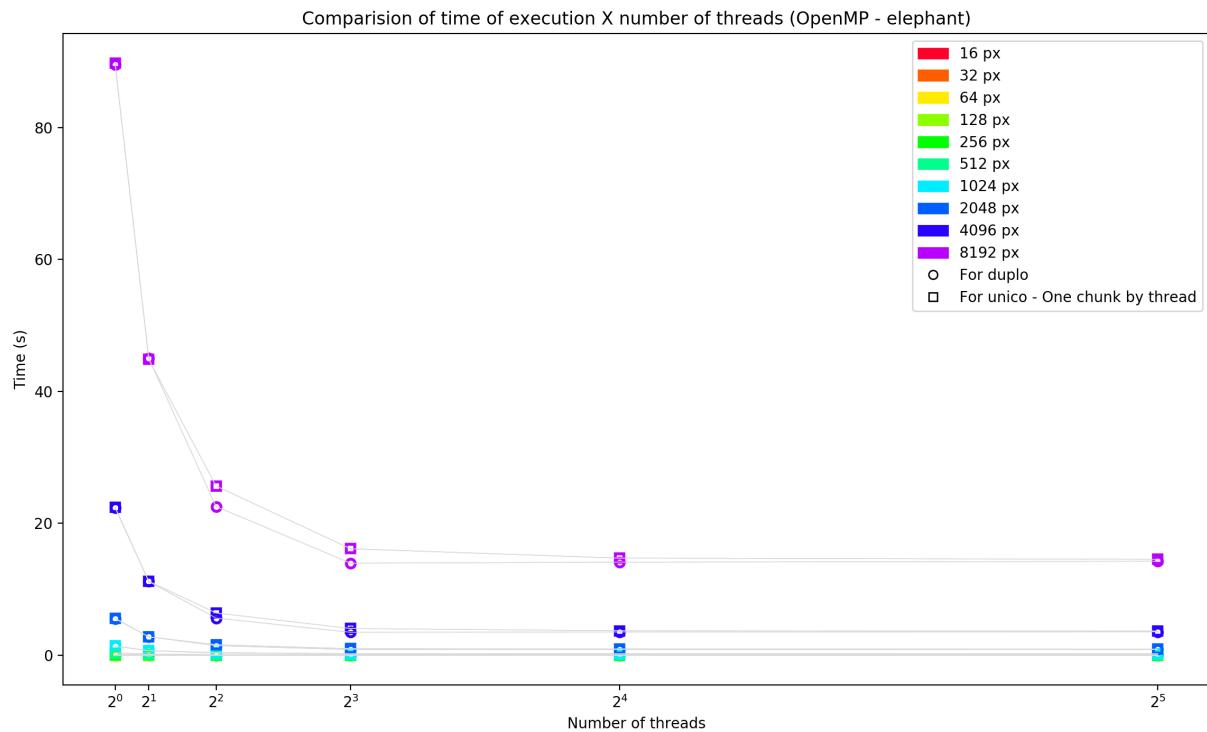
Para o caso da região full nota-se que em todos os casos a versão com dois fors foi mais rápida que a versão com somente um. Apesar do gráfico da região Full parecer que há mais variação também devemos lembrar que o eixo Y dela vai de 0 a 16, enquanto os demais ultrapassam 80. Se verificarmos as diferenças entre tempos com o mesmo número de threads, haverá sempre uma diferença semelhante em todas as regiões. Como talvez o problema sejam os chunks sejam muito pequenos e por conta disso o programa gaste muito tempo fazendo troca de contexto, o próximo experimento foi feito com chunks maiores. Dividimos o número de interações do for pelo número de threads, isto é, cada thread terá um chunk para si:

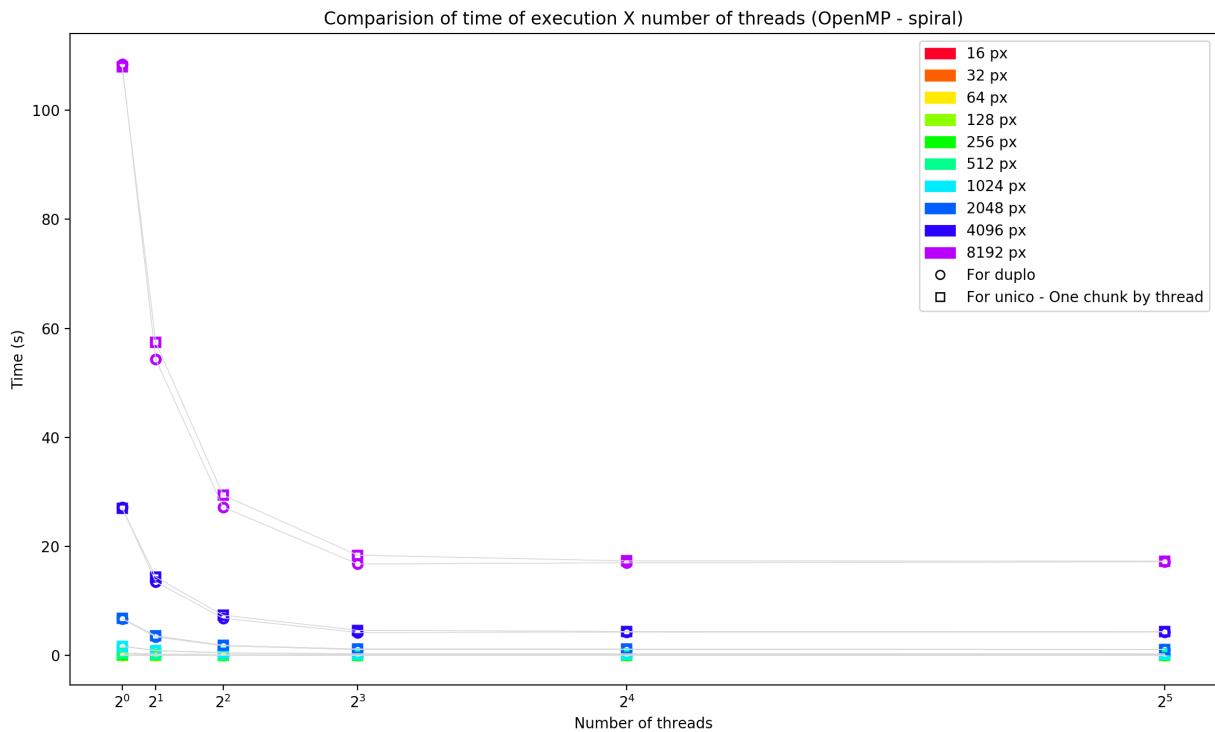
```

1 #pragma omp parallel for private(...) num_threads(nThreads) schedule(
2     dynamic, i_y_max * i_x_max / nThreads)
3
4 for (i = 0; i < i_y_max * i_x_max; i++) {
5     i_y = i / i_y_max;
6     i_x = i % i_y_max;
7
8     ...
9 }
```

No entanto os resultados ainda foram piores do que o for duplo com `schedule(dynamic)`:







Nesse caso como temos um chunk por thread, caso esse seja um chunk onde haja computações mais complexas que os demais, é provável que as demais threads terminem antes e somente uma fique processando. Nesse cenário, haveria desperdício dos recursos do computador e por isso a implementação com 2 fors, que divide os chunks em tamanhos menores, teve resultados com menor tempo de execução.

4.2 Dynamic vs Static

Realizamos também testes entre `schedule(dynamic)` e `schedule(static)`. Para cada um deles comparamos 3 tamanhos de chunk: chunks de tamanho 1, chunks de tamanho igual ao tamanho de entrada (lado da imagem) e chunks de tamanho tal que haja um chunk por thread ($tamanho_entrada^2/nThreads$)

Para chunks de tamanho tal que haja um chunk por thread, notamos que a implementação estatica e dinâmica executam em tempos quase identicos:

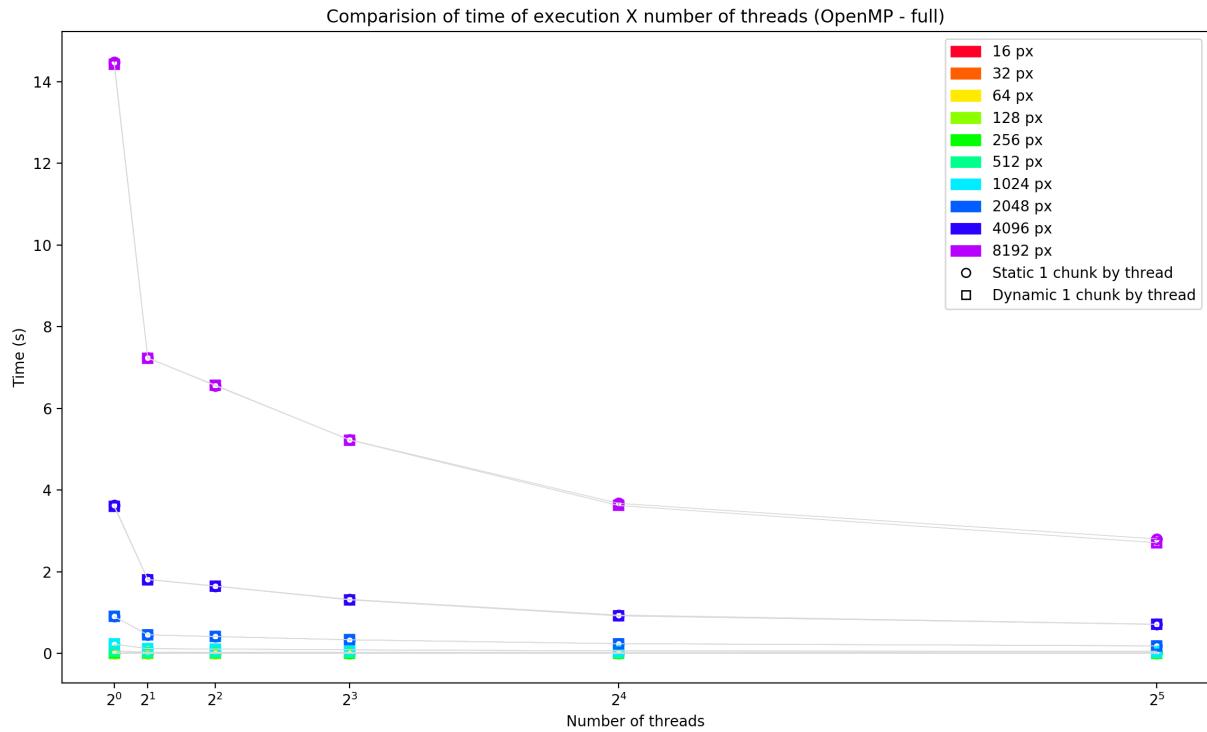


Figura 8: Grafico da região full

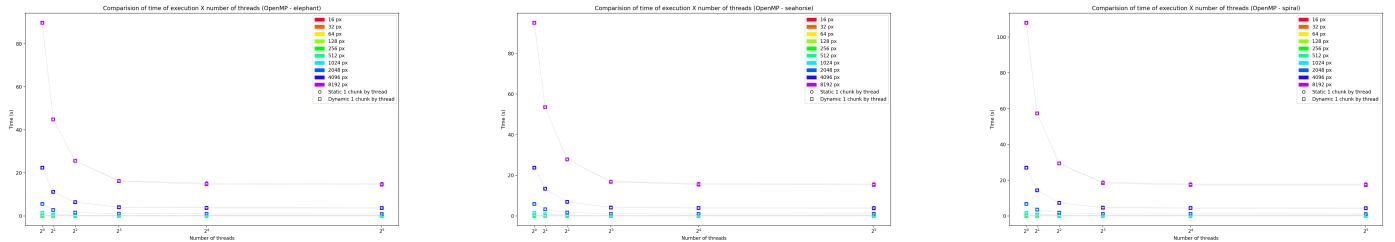


Figura 9: Graficos das regiões Elephant, Seahorse e Spiral

Para chunks de tamanho igual ao tamanho de entrada, também notamos que as implementações estatica e dinâmica executam em tempo bastante semelhantes:

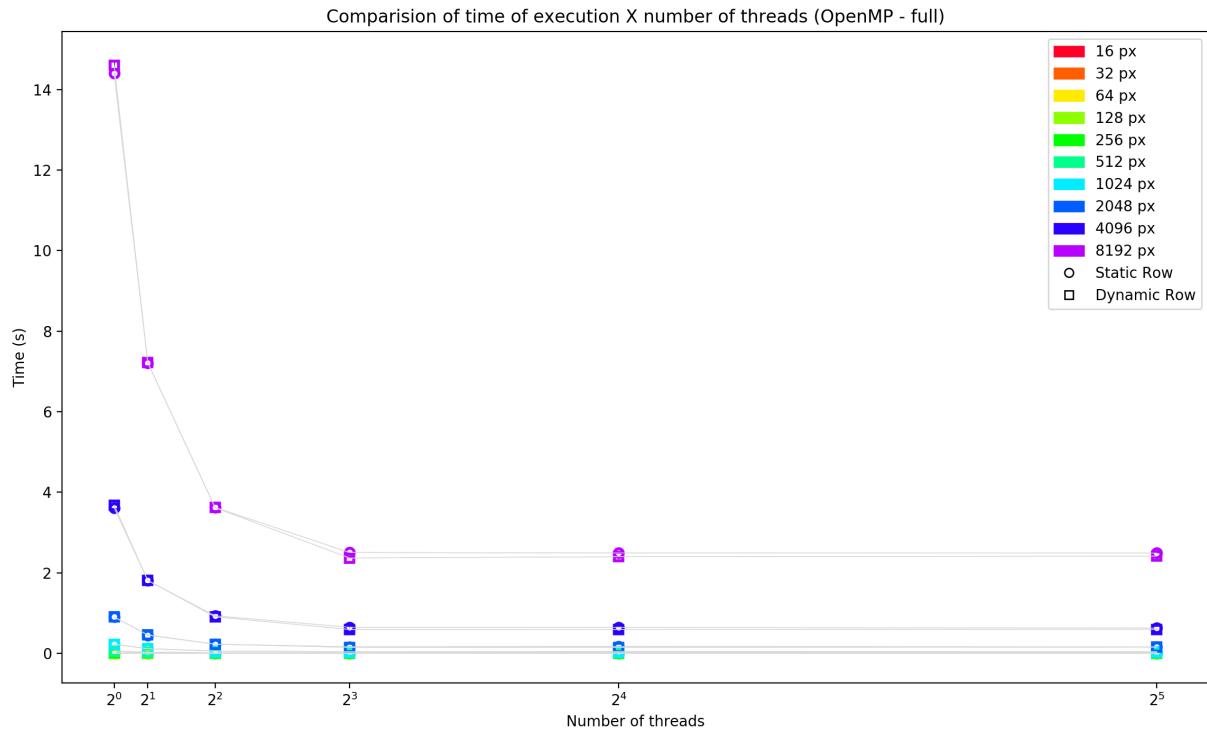


Figura 10: Grafico da região full

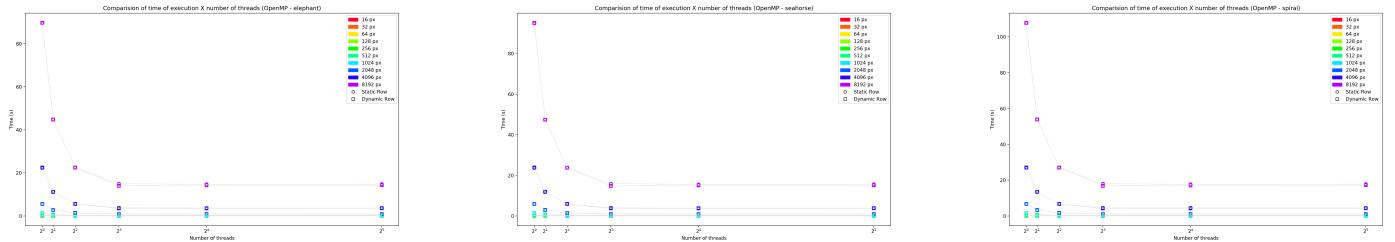
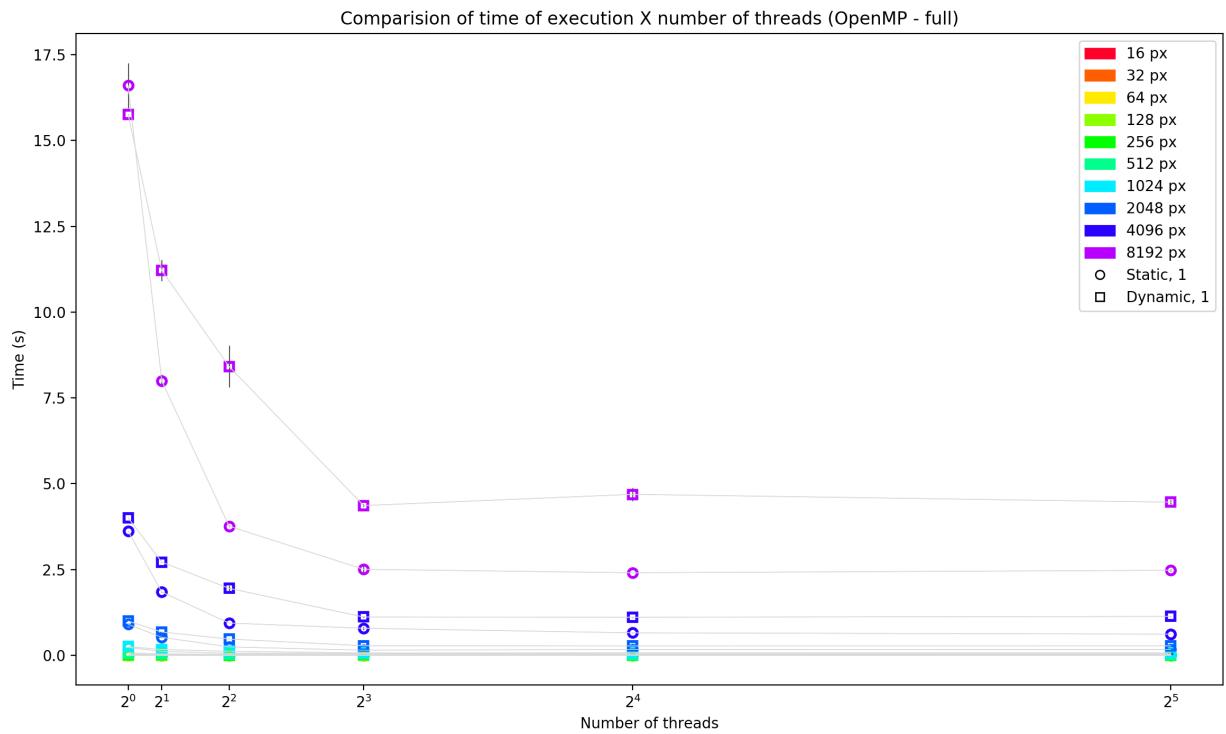


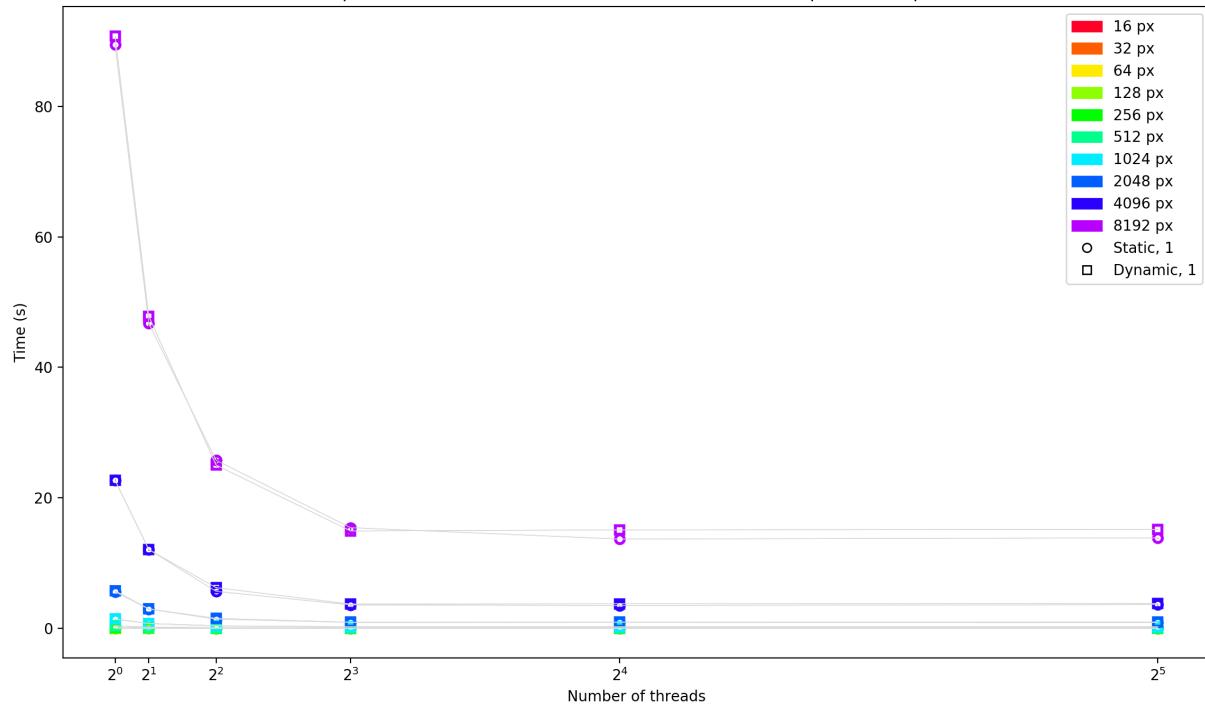
Figura 11: Graficos das regiões Elephant, Seahorse e Spiral

Para chunks de tamanho 1, obtivemos valores mais variados:

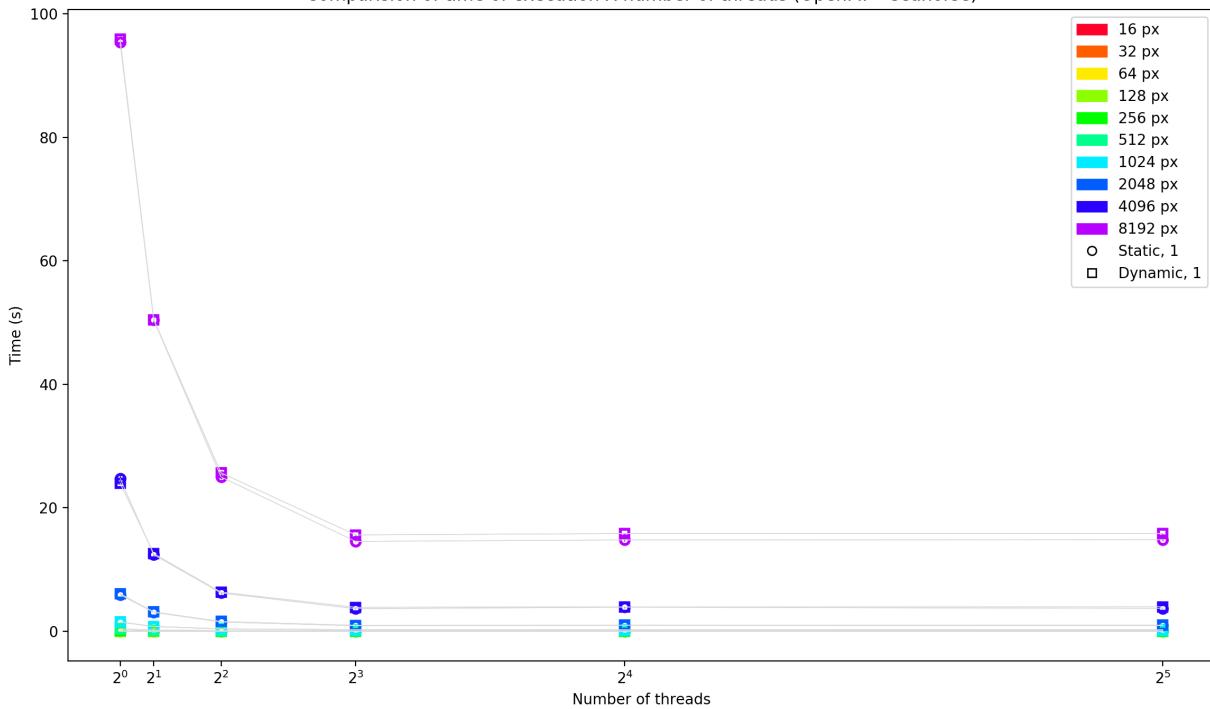


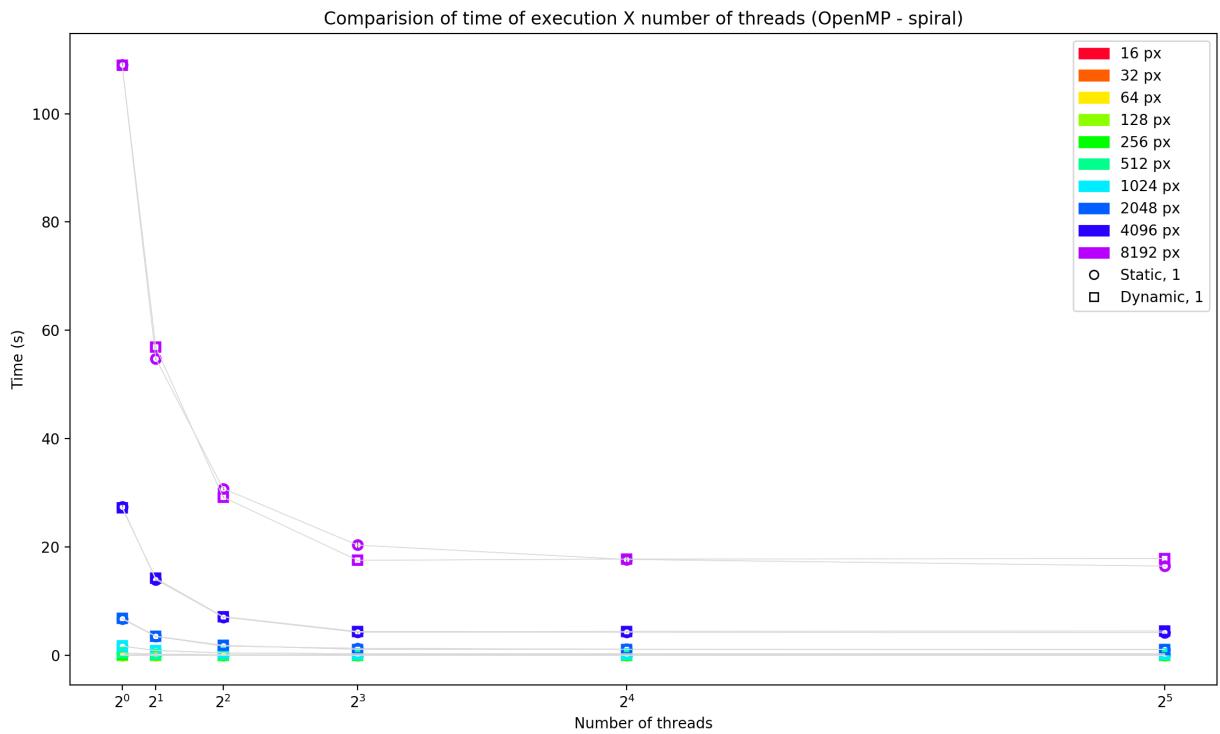
Vemos que nesse caso a implementação estática se comportou melhor que a dinâmica. No entanto para regiões com mais pontos de muitas iterações do cálculo de mandelbrot notamos que para 4 e 8 threads a implementação dinâmica apresenta resultados menores que a estática:

Comparision of time of execution X number of threads (OpenMP - elephant)



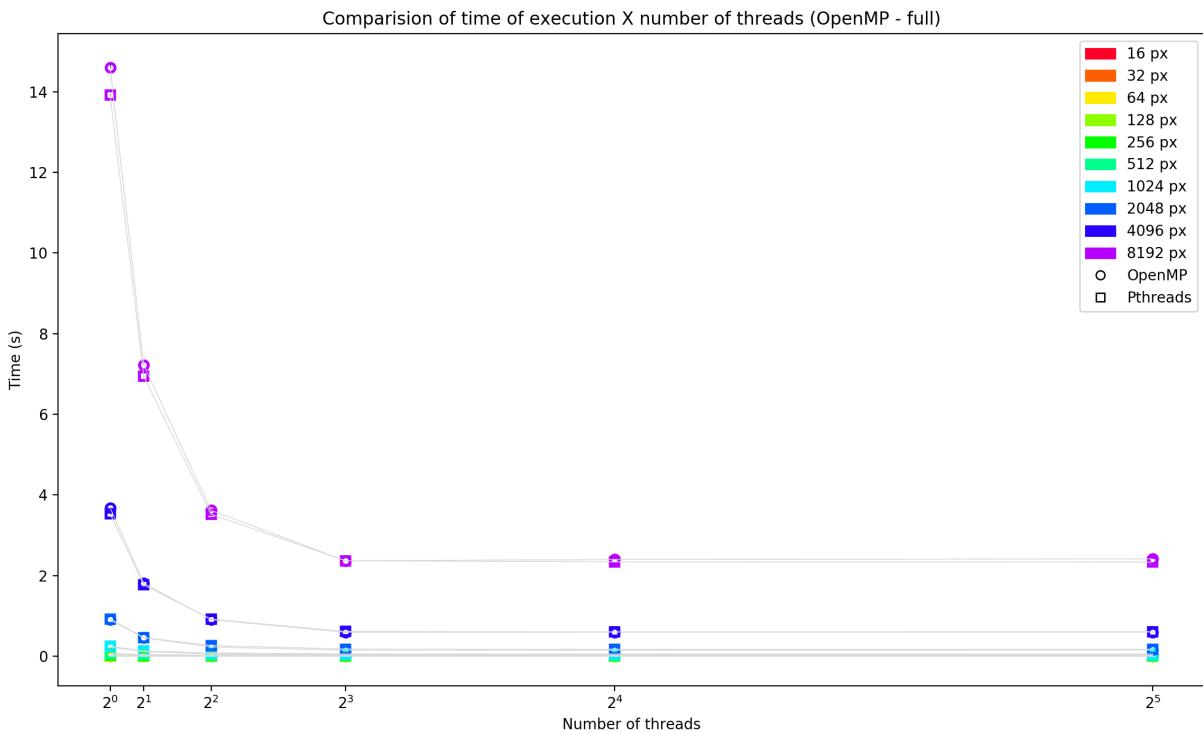
Comparision of time of execution X number of threads (OpenMP - seahorse)



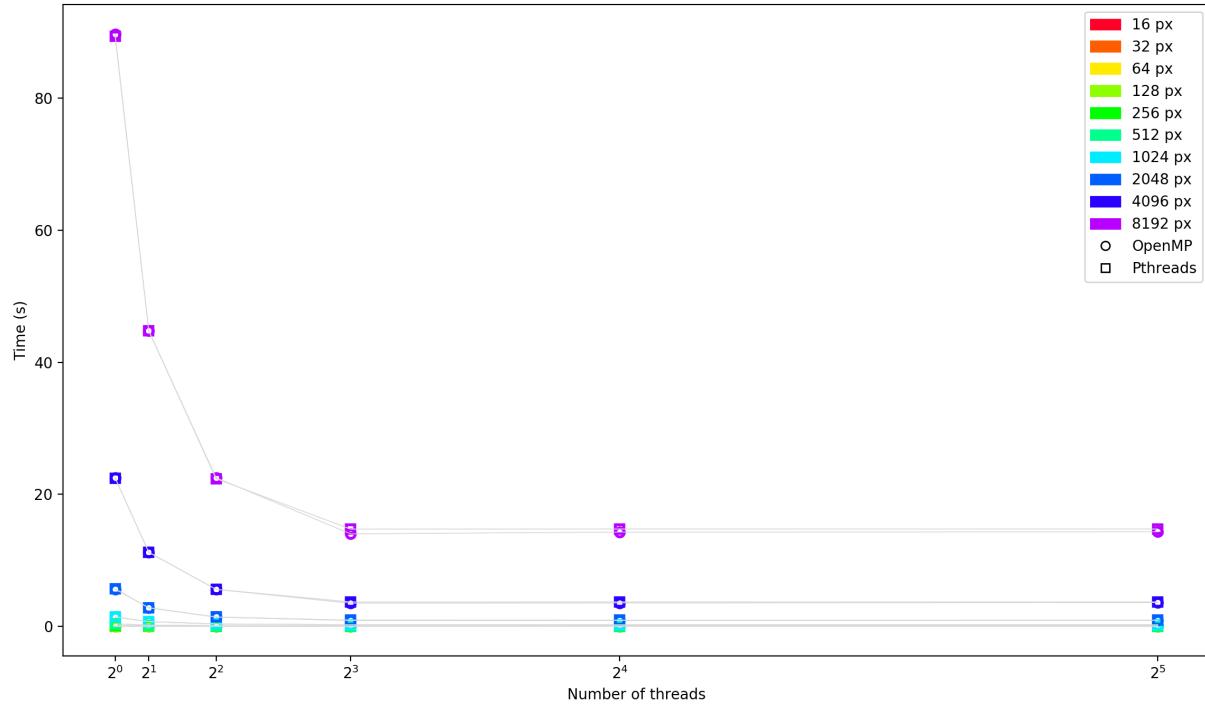


5 OpenMP vs Pthreads

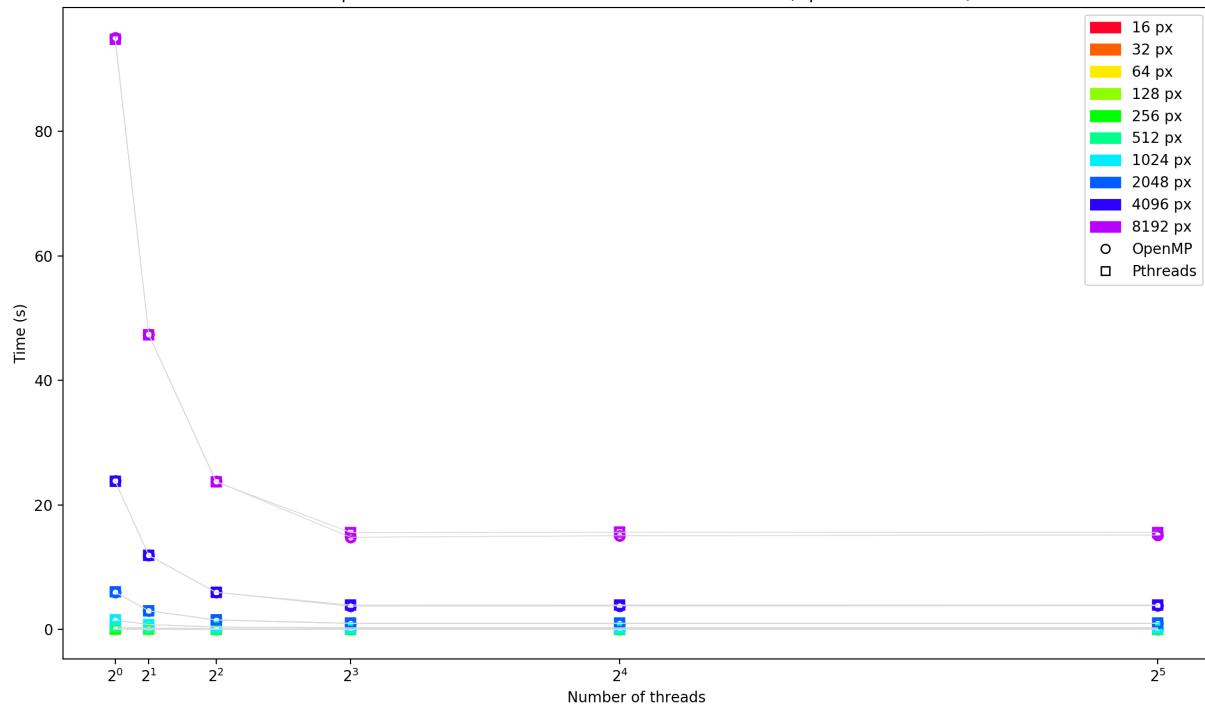
Vimos que para ambos, OpenMP e Pthreads, a melhor opção de implementação foi utilizando escalonamento dinâmico e tamanho de chunk igual ao tamanho de entrada. Comparando esses dois algoritmos obtivemos os seguintes resultados:

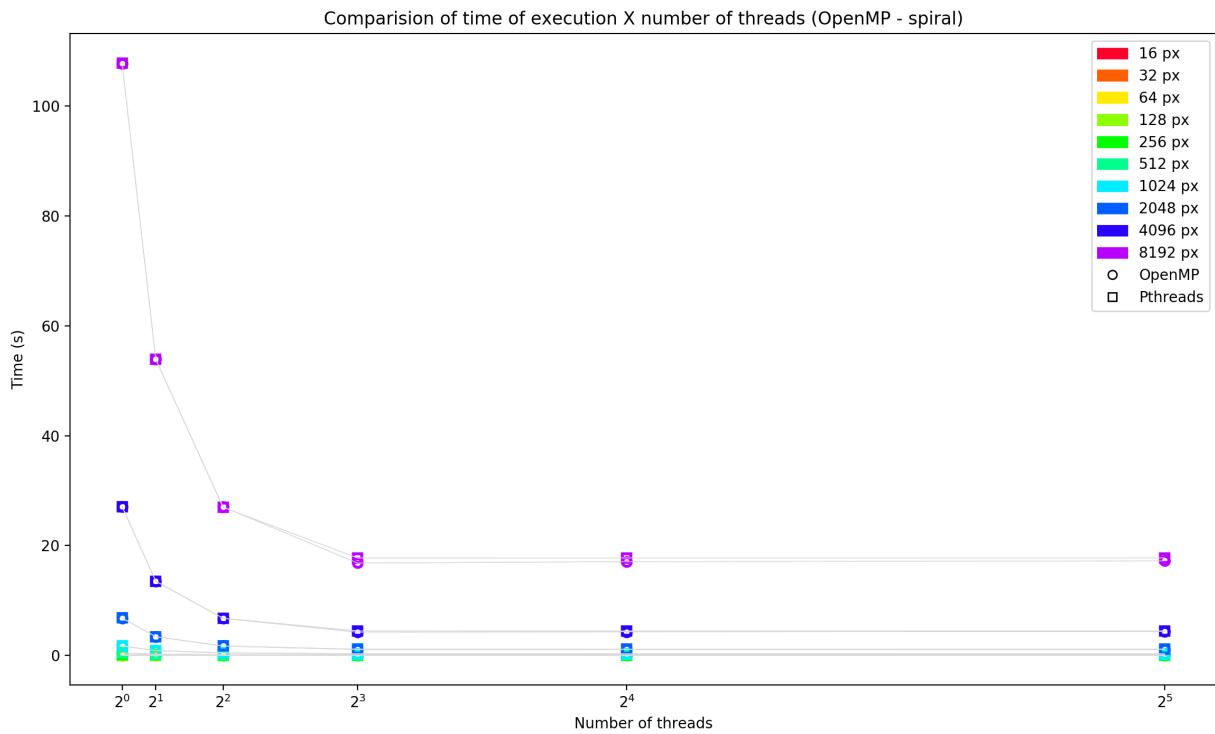


Comparision of time of execution X number of threads (OpenMP - elephant)



Comparision of time of execution X number of threads (OpenMP - seahorse)





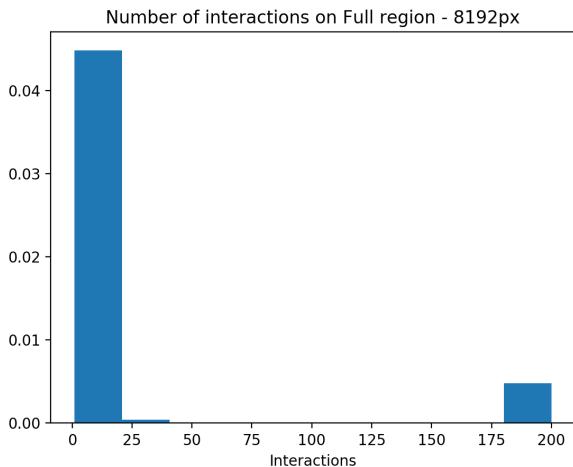
Analizando esses resultados nota-se que as duas implementações apresentam quase os mesmos resultados. Para esse problema vimos que o OpenMP nos fornece o mesmo desempenho que uma implementação em Pthreads com a facilidade de só ter que adicionar uma linha ao código legado.

6 Discussão dos Resultados

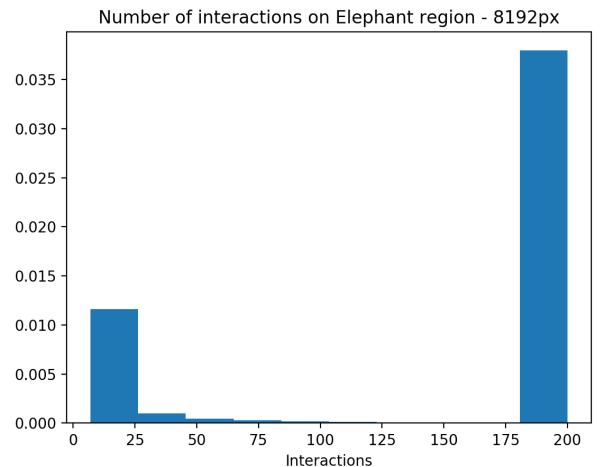
Por que é recomendado realizar mais de uma medição? Porque os resultados observados dependem do estado da máquina durante a execução do programa. Como testamos nossos programas em máquinas com um sistema operacional, rodando diversos outros programas ao mesmo tempo, é esperado que fatores como fila de processos, paginação de memória, etc, interfira no tempo de execução do nosso programa. Portanto, se tiramos uma média de várias execuções, somos capazes de dizer aproximadamente o comportamento médio do programa.

Como e por que as três versões do programa se comportam com a variação: Do tamanho da entrada? Das regiões do Conjunto de Mandelbrot? Do número de threads? As variações de tempo de execução em relação ao tamanho da entrada ocorrem, pois o tamanho do `for` dentro da função `compute_mandelbrot()` depende do tamanho da entrada. Isto significa que o `for` será executado mais vezes se o tamanho da entrada for maior e consequentemente levará mais tempo para executar o programa.

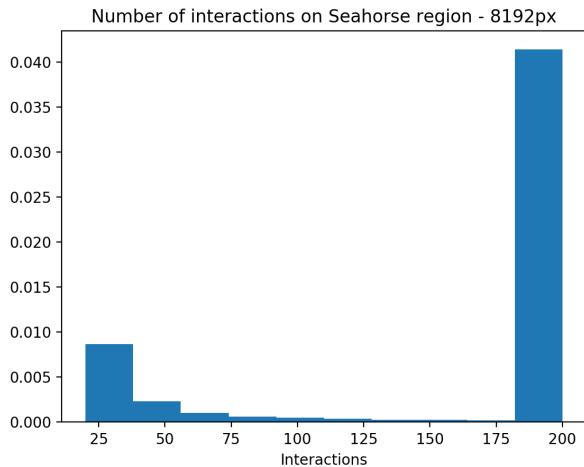
As variações causadas pelas diferentes regiões são devido ao maior número de pontos onde há maior número de iterações. Vejamos os histogramas de cada uma das regiões:



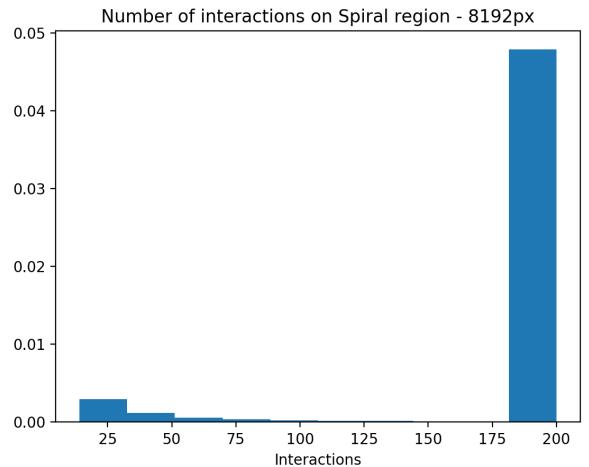
(a)



(b)



(c)



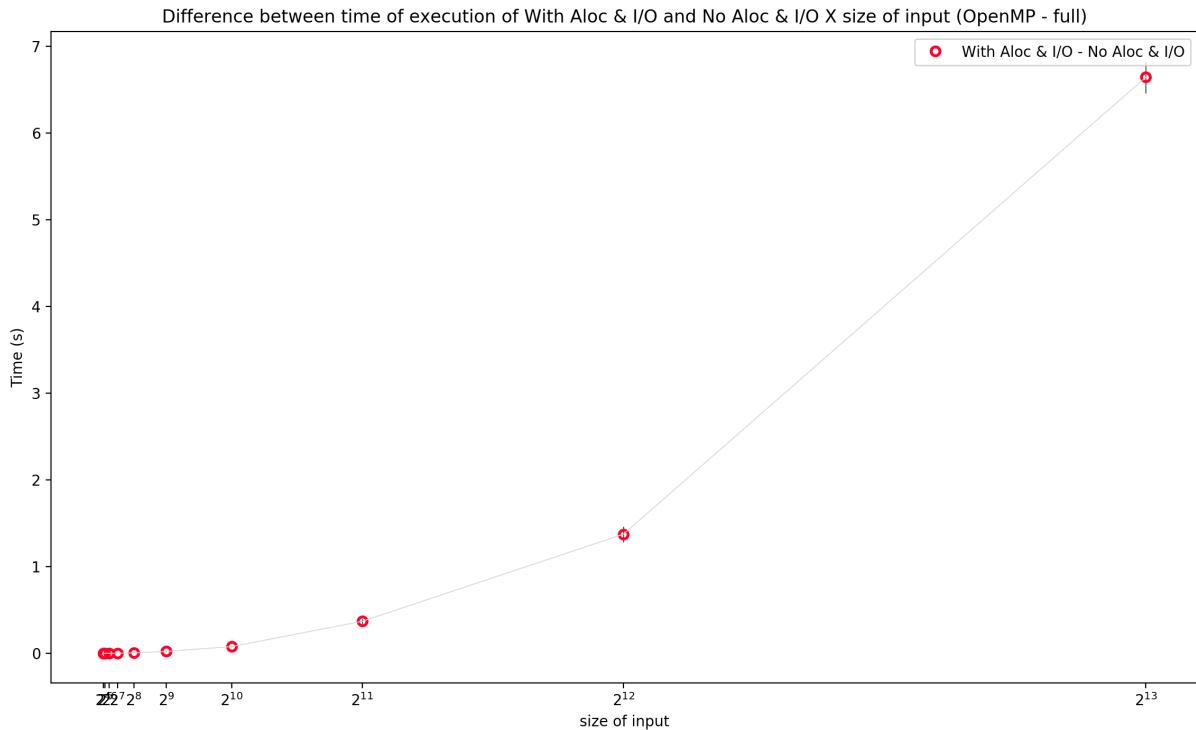
(d)

Na região Full vemos que a maior parte dos pontos tem poucas iterações do calculo de mandelbrot. A média $\mu = 22.038$ e o desvio padrão $\sigma = 58.217$ comprovam esse fato. Tais valore comparados aos da região Elephant ($\mu = 151.757$, $\sigma = 80.798$), Seahorse ($\mu = 160.667$, $\sigma = 69.371$) e Spiral ($\mu = 183.204$, $\sigma = 49.430$) nos mostram porquê os calculos das regiões, Elephant, Seahorse e Spiral levaram mais tempo. Como nessas regiões há mais iterações em geral, mais tempo é gasto calculando. Isso é multiplicado por cada ponto, o que causa um bom atraso na execução do programa.

Quanto ao número de threads, é esperado que um gráfico de tempo de execução por número de threads tenha a forma de um U. Isso acontece, por uma lado, porque quando temos

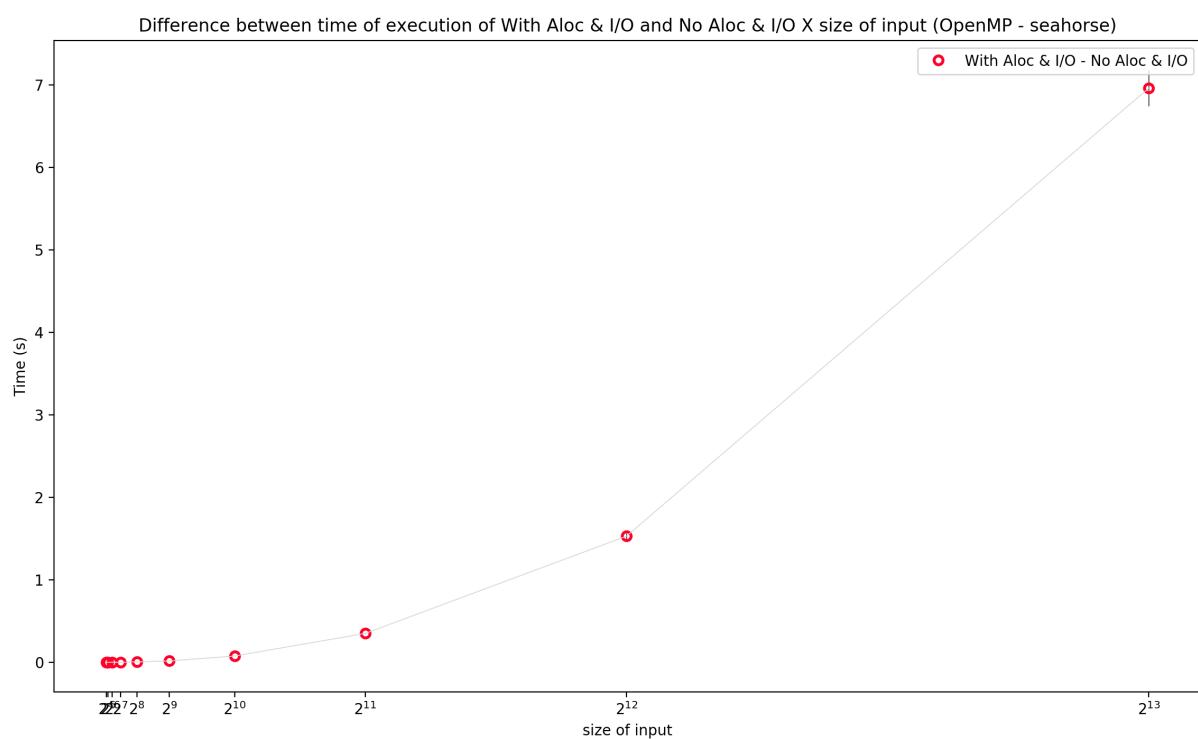
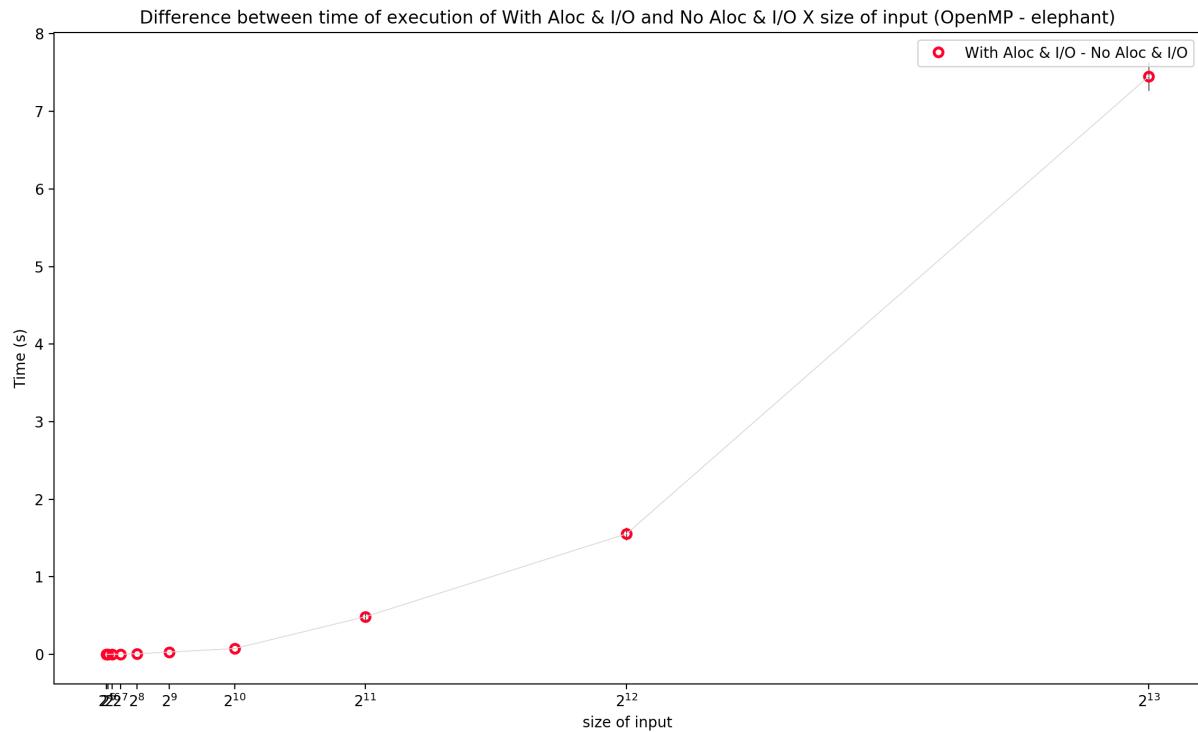
poucas threads não usamos o número total de processadores da máquina, logo aumentar o número de threads aumenta o desempenho. Obviamente, essa melhoria é saturada em uma máquina com um número finito de processadores, e além disso, o número grande de threads exige um maior controle, seja do programa principal ou do escalonador do sistema operacional, explicando um aumento no tempo de execução quando aumentamos o número de threads.

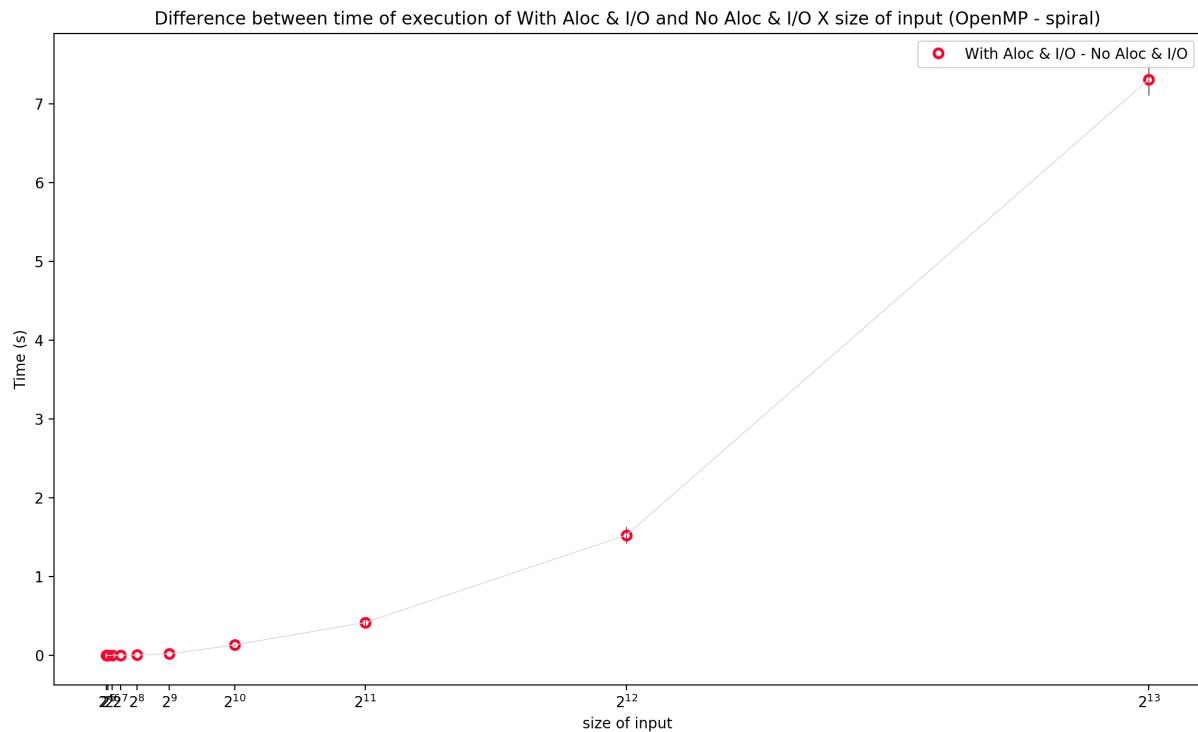
Qual o impacto das operações de I/O e alocação de memória no tempo de execução? Vimos na análise da implementação sequencial que as operações de leitura e escrita e a alocação da memória afetam consideravelmente o desempenho. Gostaríamos de ver qual foi a diferença entre a implementação com operações de I/O e com alocação em relação à sem ambos. Se pudermos tomar essa diferença teremos uma estimativa do tempo que leva as operações I/O e alocação:



Nota-se que o tempo levado pelas operações de I/O e alocação dinâmica tem comportamento exponencial em relação ao tamanho da entrada. Vale lembrar que o tamanho da entrada é somente o lado da imagem, portanto o tamanho da matriz a ser alocada também é exponencial em relação ao tamanho da entrada. Para as demais regiões nota-se um com-

portamento semelhante:





Notemos ainda que os valores também são semelhantes, independente da região. Como explicado anteriormente, isso se deve-se ao fato de que as operações de I/O e a alocação de memória independem do número de iterações do cálculo de mandelbrot em cada ponto.

7 Conclusão

As discussões que tivemos ao longo do trabalho, enquanto procuramos deixar o código mais rápido, foram parecidas para ambas implementações, com Pthreads e OpenMP. A principal delas foi decidir entre utilizar uma estratégia estática ou dinâmica ao paralelizar um laço, o que nos levou a entender melhor a paralelização do código e perceber que, acima de tudo, os melhores resultados são atingidos quando dividimos o trabalho igualmente entre as threads. Esse problema parece óbvio se não considerarmos que, como no problema enfrentado, nem todo pedaço de mesmo tamanho do problema consumirá o mesmo tempo de execução.

Outra discussão interessante que foi levantada nesse trabalho, na implementação com pthreads é a responsabilidade de escalar um pedaço de trabalho. Temos duas opções nesse caso: cuidar da divisão do trabalho ou jogar o problema para um nível abaixo, no sistema operacional. A primeira solução torna necessário fazer todo o controle de divisão de trabalho em uma threads principal e criar poucas threads (geralmente o mesmo número de processadores na máquina para se resolver o problema). Na segunda opção, criamos uma thread para cada pedaço a ser resolvido, mesmo que isso torne o número de threads muito grande, e deixamos com o sistema operacional a responsabilidade de escalar todos esses pedaços que precisam ser resolvidos.

Os testes feitos mostraram que o desempenho do código implementado em OpenMP e PThread foram bem parecidos. Porém, gostaríamos de destacar que a implementação do código em OpenMP foi muito mais rápida do que a implementação em PThreads. Portanto, especificamente para o problema do EP, OpenMP parece ser uma ferramenta mais adequada.

Entretanto, enquanto analisávamos os dados coletados em nossos programas, na maioria das vezes que obtemos resultados inesperados, a biblioteca que estava sendo usada era OpenMP, o que se deve ao fato de não termos implementado de fato uma paralelização; apenas indicamos ao compilador como o código deve ser paralelizado. Na implementação com pthreads, por outro lado, temos melhor ideia de como o código será executado e temos maior liberdade para decidir sobre isso. Imaginamos que, em problemas mais complexos e com concorrência mais complicada, é possível que um código em Pthreads seja mais rápido do que um código em OpenMP.