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

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

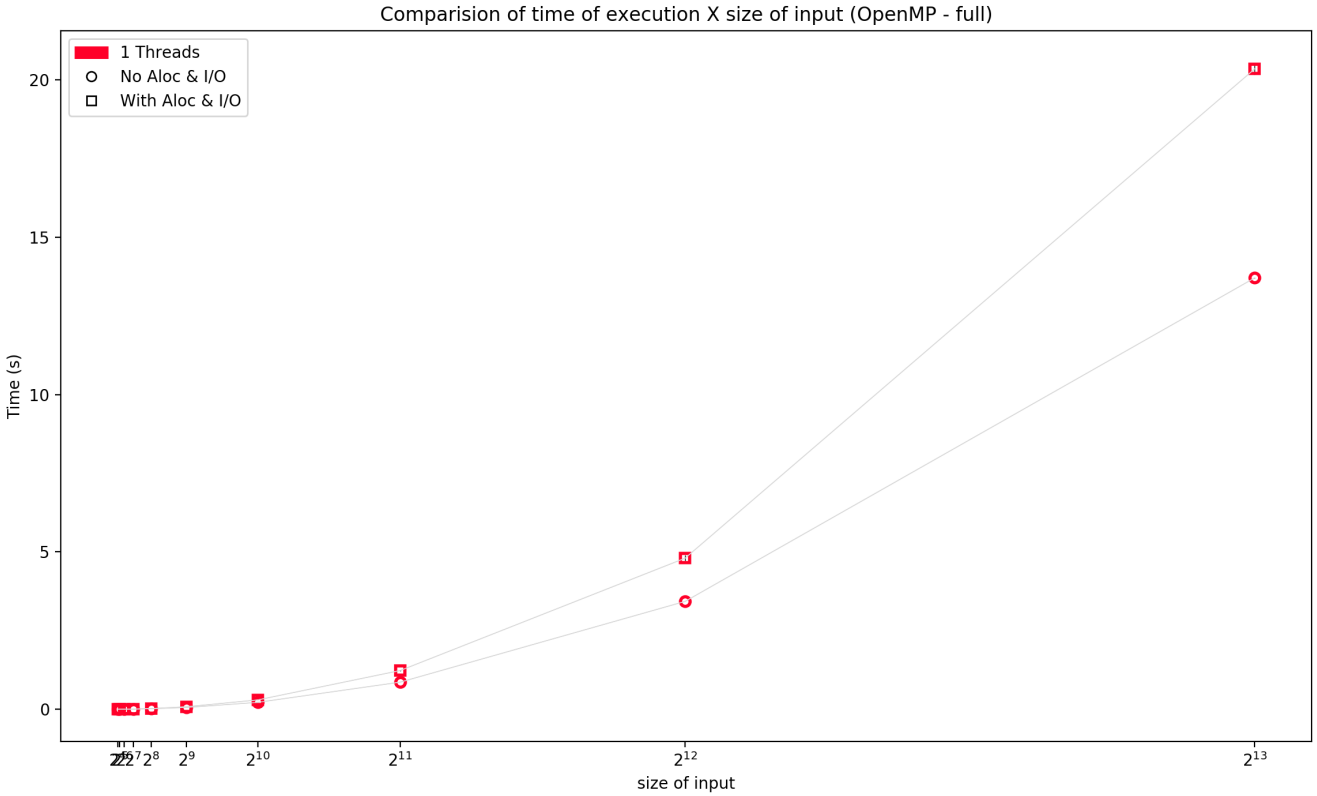
Ao longo desse trabalho, iremos apresentar resultados de tempo de execução das diferentes implementações e suas respectivas variações. Para isso, utilizamos a ferramenta *perf*, capaz de realizar repetições de experimentos, apresentando resultados médios e com desvio padrão. Todos os resultados apresentados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

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.

2 Código Sequencial

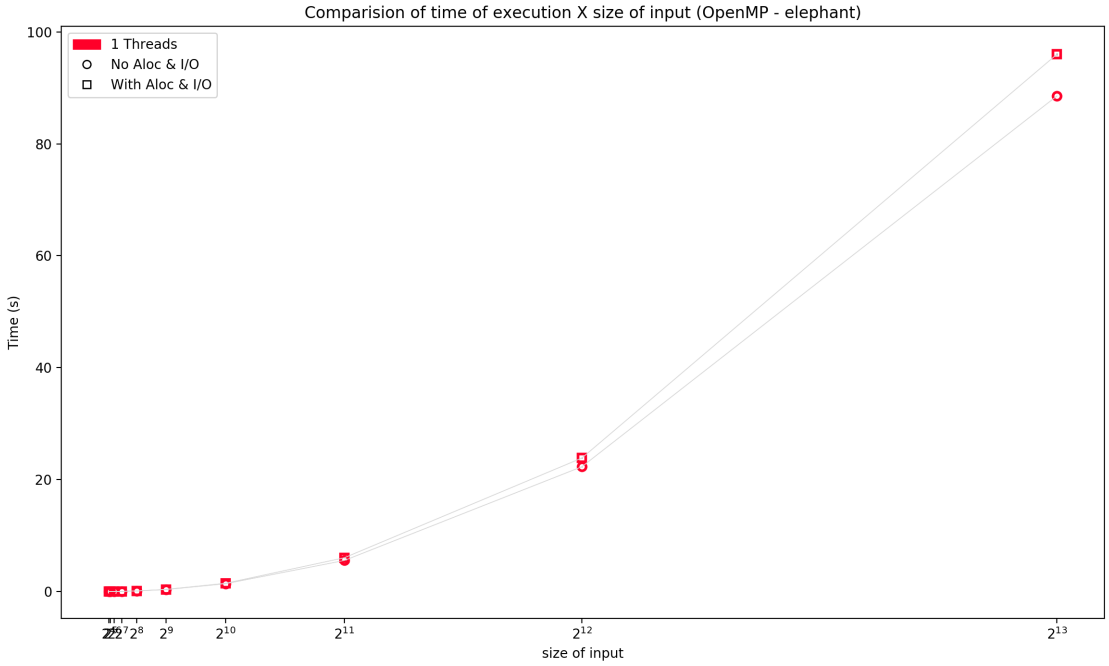
Para implementar a versão sequencial do programa do cálculo do conjunto de Mandelbrot temos uma versão com alocação de memória e com comandos de leitura e escrita e uma versão sem ambos. No caso sem ambos, como não há comandos de leitura, os comandos de leitura e escrita são retirados, removendo-se a 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 substituimos a alocação dinâmica do `image_buffer` por uma alocação estática: `char image_buffer[11500*11500][3]`.

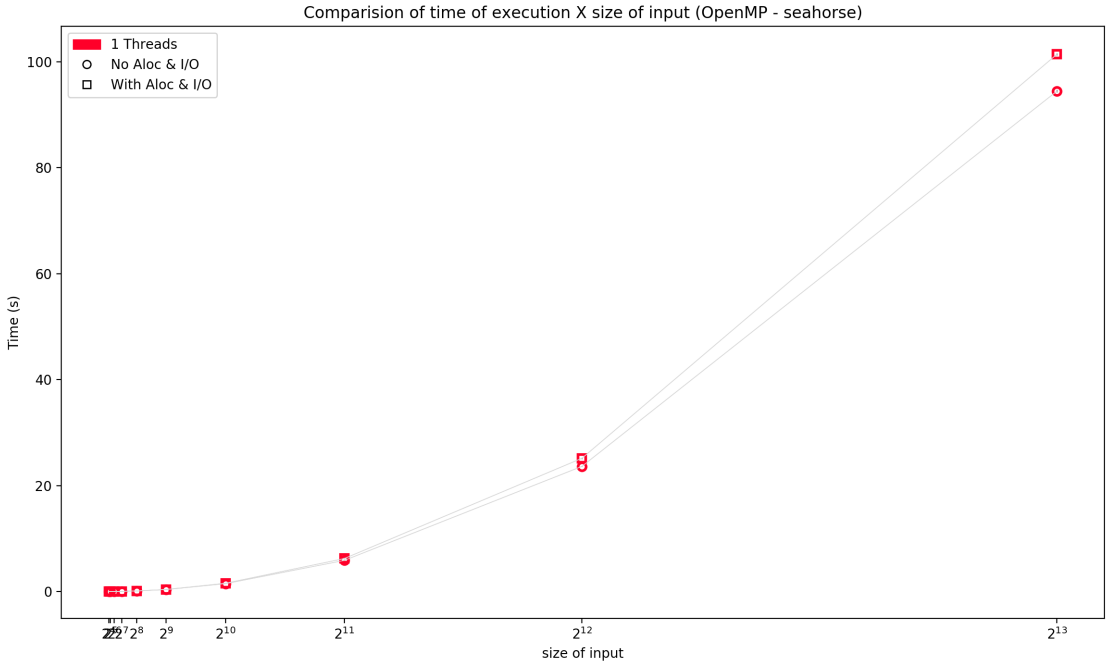
Comparemos os resultados obtidos nos testes com e sem alocação de memória e comandos de leitura e escrita:

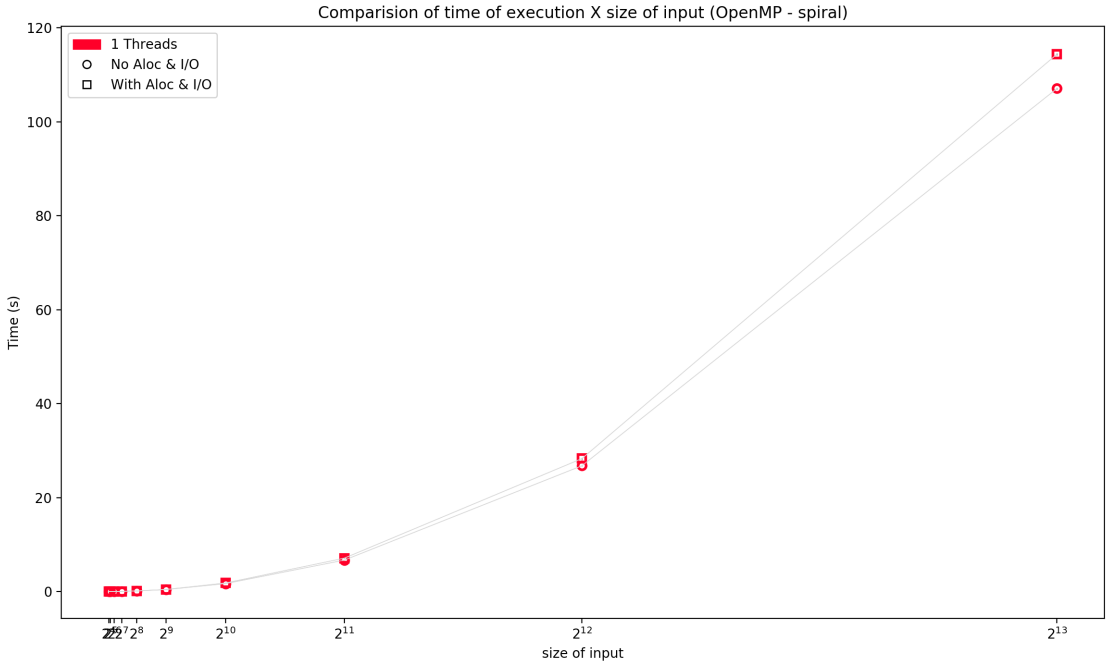


Como esperado o tempo do programa sem alocação e sem comandos de leitura e escrita é

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

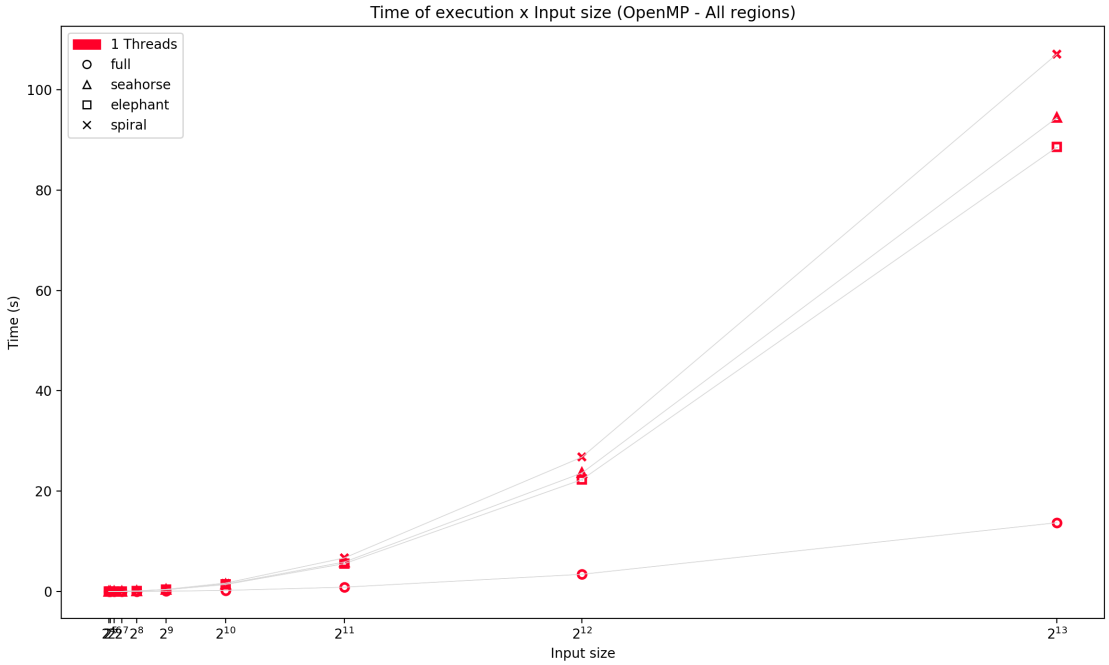


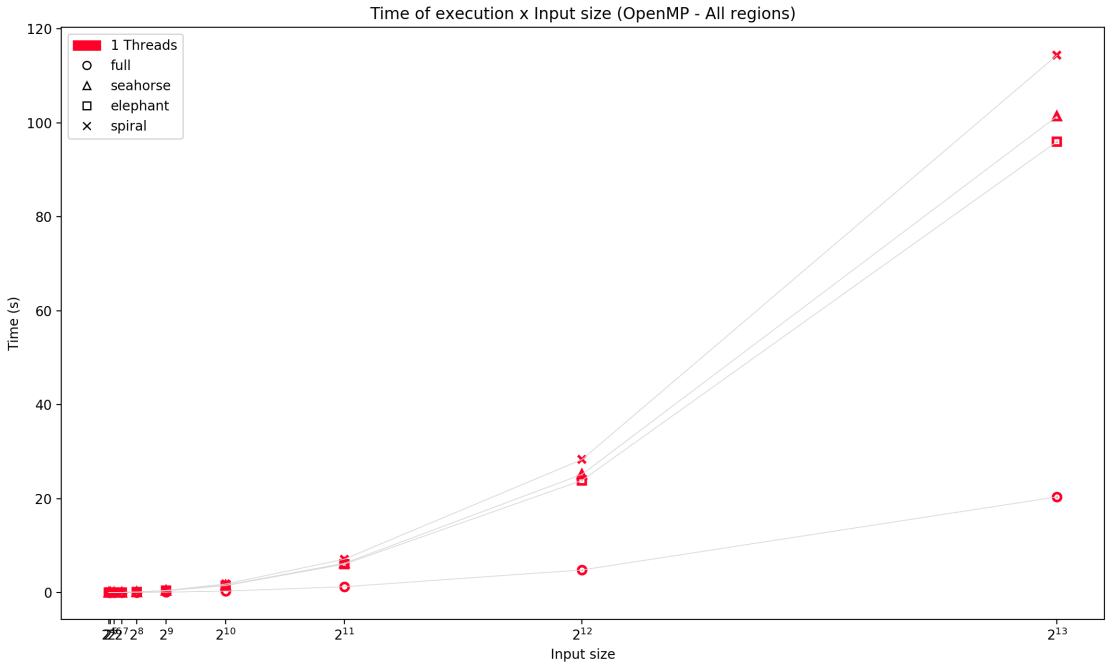




Nota-se que o tempo em cada região é diferente, pois em cada região pode haver mais cálculos de pontos com mais interações. Notemos ainda que a diferença de tempo para cada tamanho de entrada se mantem mesmo mudando-se as regiões. Por exemplo, a versão sem alocação dinamica e sem operações de leitura e escrita executa cerca de 10 segundos mais rapido que a versão com, dado a entrada de tamanho $2^(13)$, independente da região. Isso ocorre, pois comandos de leitura e escrita e alocação dinamica, 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:


Time of execution x Input size (OpenMP - All regions)
1 Threads
full
seahorse
elephant
spiral
Time (s)
Input size


Time of execution x Input size (OpenMP - All regions)
1 Threads
full
seahorse
elephant
spiral
Time (s)
Input size

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

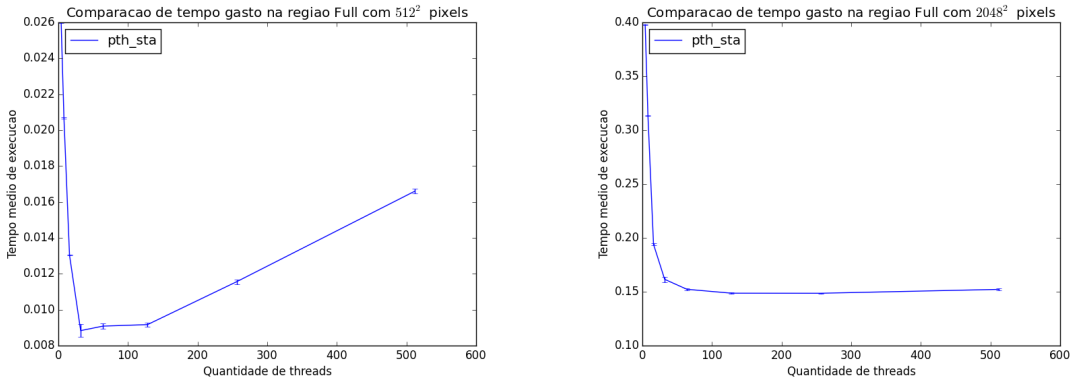
Chamamos de implementação com divisão estática a versão do nosso código em pthreads no qual o trabalho é dividido em n pedaços de mesmo tamanho, e cada pedaço é dado a uma thread. Chamamos essa implementação de estática porque cada thread recebe apenas um bloco de trabalho, pré-determinado pela divisão 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.



(a) (b)

Figura 1: É possível notar em ambas figuras que o aumento do número de threads de fato diminui o tempo de execução do programa. No gráfico 1(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 ← lista de nacos de todos os pixels
3:     while S ≠ ∅ do
4:         chunk ← S.popChunk ()
5:         Espere alguma thread estar livre
6:         for all Threat 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.

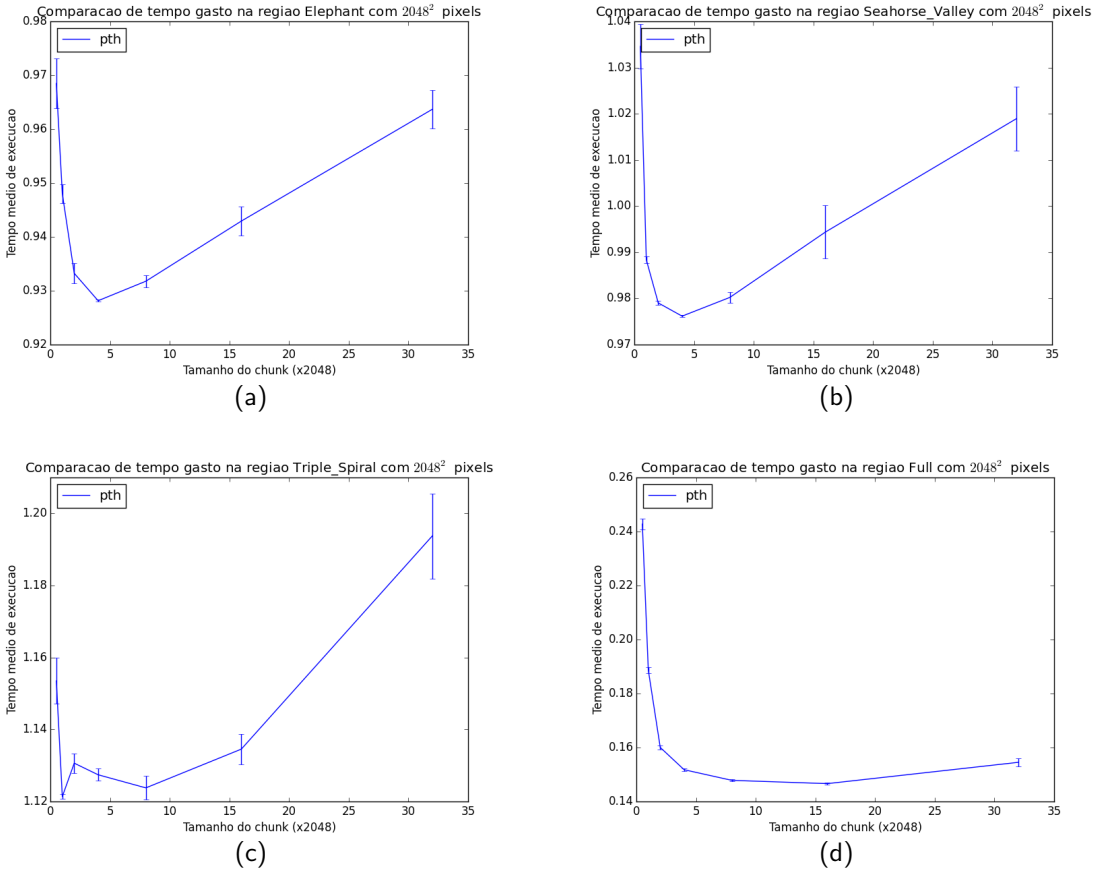


Figura 2: Verificamos empiricamente que definir o tamanho do chunk como o tamanho do lado da imagem traz bons resultados para o tempo de execução.

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.

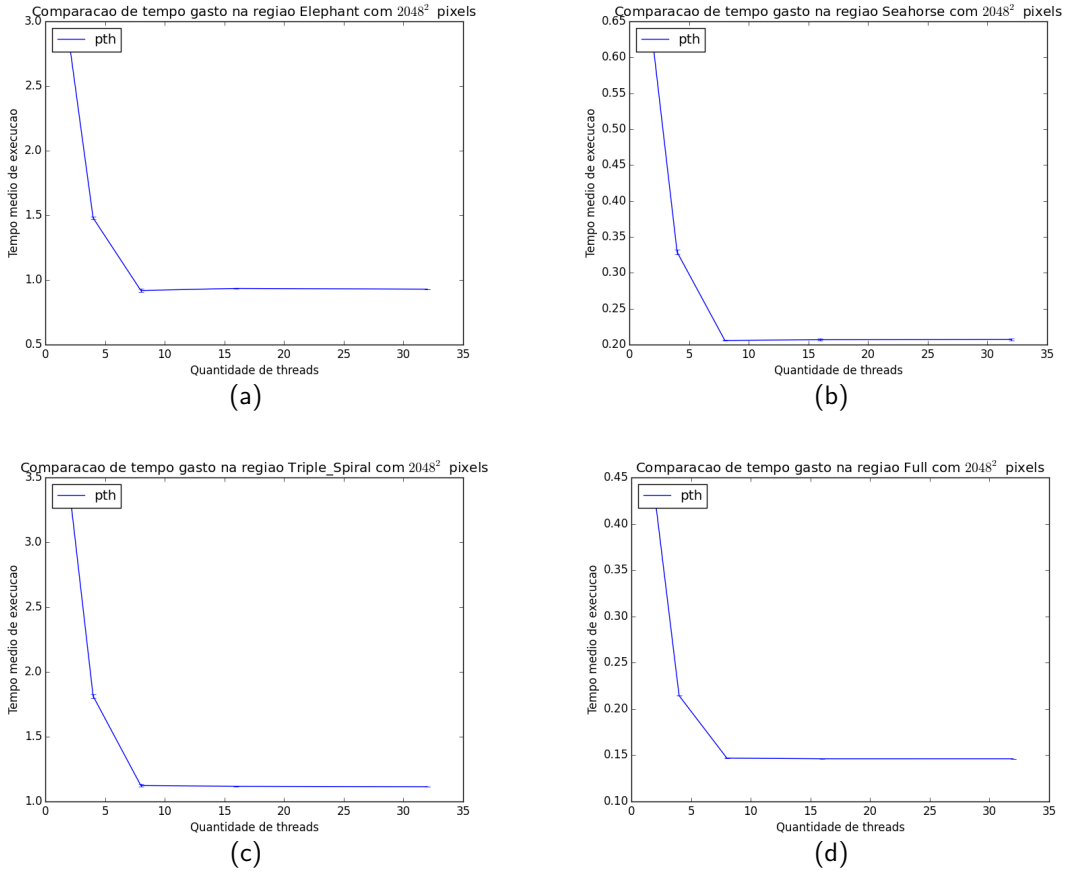


Figura 3: Podemos observar que existe uma saturação de melhora de desempenho quando o número de threads atinge o número de cores (8) da máquina.

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 1. 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 3

Veja abaixo a comparação em consumo de tempo entre as duas implementações de para-

lelismo com Pthreads:

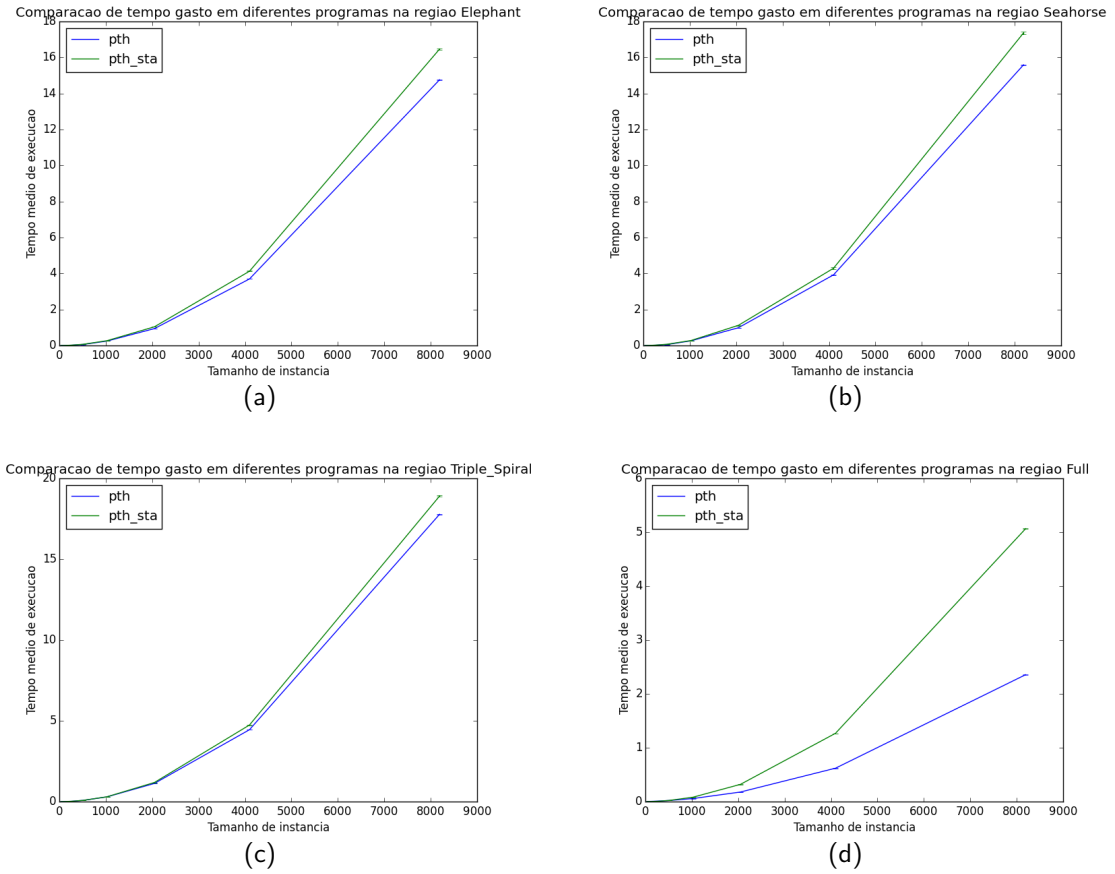


Figura 4: Podemos observar nessas figuras que a implementação com divisão dinâmica possui desempenho melhor do que estático e que, além disso, essa diferença tende a aumentar quando o tamanho da instância também aumenta. É provavel que as regiões onde o trabalho está Esses testes foram realizados com número de threads igual a quantidade de processadores da máquina.

Mas o que acontece quando rodamos a versão estática com muitas threads e comparamos com a versão dinâmica? Para responder essa dúvida, decidimos comparar as duas implementações em pthreads de maneira que o número de *chunks* fosse igual em ambas. Para o código dinâmico, o número de threads se manteve igual ao número de cores da máquina enquanto que para o código estático o número de threads passou a ser igual a quantidade de pixels em uma linha da imagem. Note que enquanto a primeira estratégia cuida do escalonamento dos chunks para cada thread, a segunda cria uma thread para cada chunk e joga a responsabilidade para baixo, com o sistema operacional.

O que observamos nesse experimento, apresentado na 5, é que o desempenho de ambas estratégias foi muito parecido, com desempenho melhor do código estático em instâncias de

tamanho perto de 2^{13}. O motivo da implementação dinâmica ter desempenho pior se deve provavelmente ao nosso escalonador em pthreads, que precisa criar e dar join em threads várias vezes. Poderíamos fazer outra implementação onde a própria thread "escolhe"um chunk para calcular, mas não o faremos por falta de tempo. Apesar disso, é visível que esse overhead se torna insignificante conforme aumentamos o tamanho da instância sendo resolvida.

Por fim, consideramos melhor a implementação dinâmica, porque ela depende menos do sistema operacional e máquina utilizada. Muitos sistemas operacionais definem limite na quantidade de processos na máquina, o que traz um limite no programa estático. Além disso, caso o escalonador do sistema usado seja muito ruim, o desempenho do código dinâmico deve ser muito melhor.

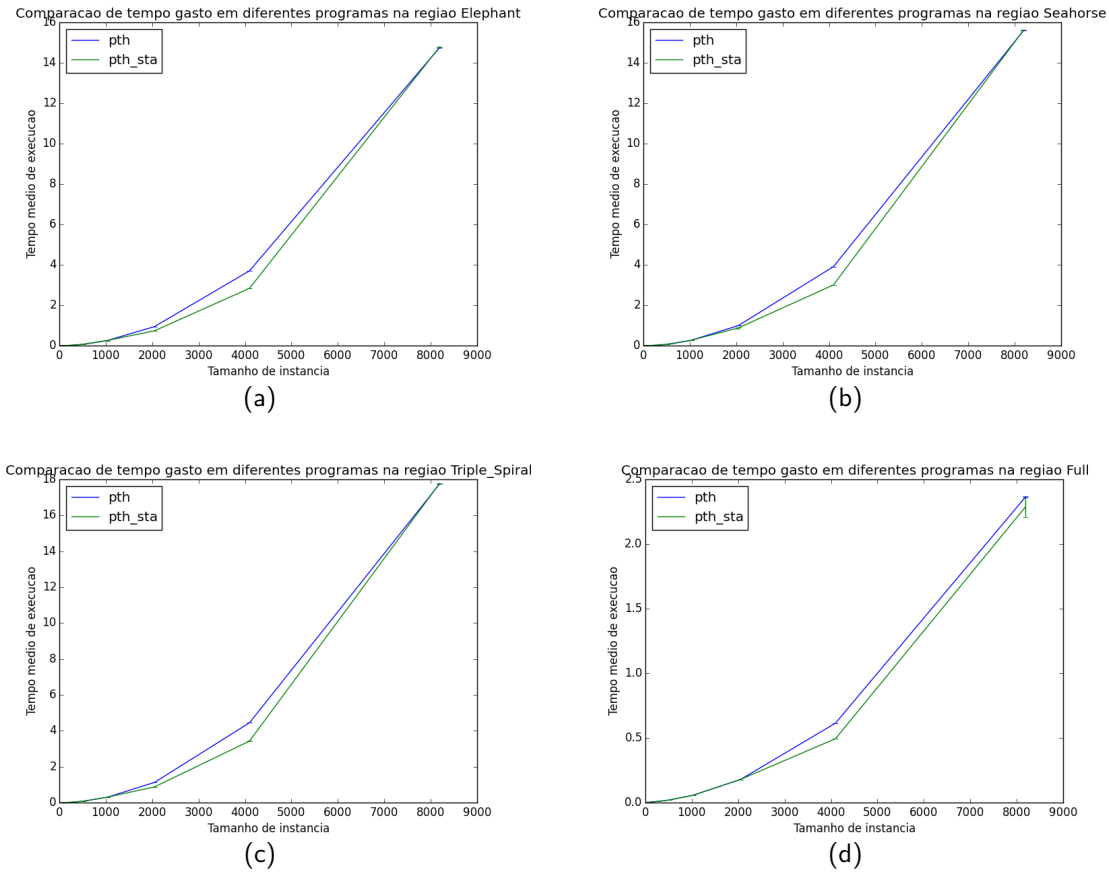


Figura 5: Neste teste, o número de threads usada na implementação estática foi igual ao de pixels em uma linha da imagem. No código dinâmico, o número de threads foi exatamente o número de processadores da máquina, além disso, a quantidade de chunks é igual ao número de pixels em uma linha da imagem. Lembrando que o código estático cria um número de chunks igual ao de threads, podemos dizer que o número de chunks é igual em ambas implementações.

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(
   dynamic)
2  for (i_y = 0; i_y < i_y_max; i_y++) {
3      c_y = c_y_min + i_y * pixel_height;
4      ...
```

Implementando dessa forma obtemos os seguintes resultados:

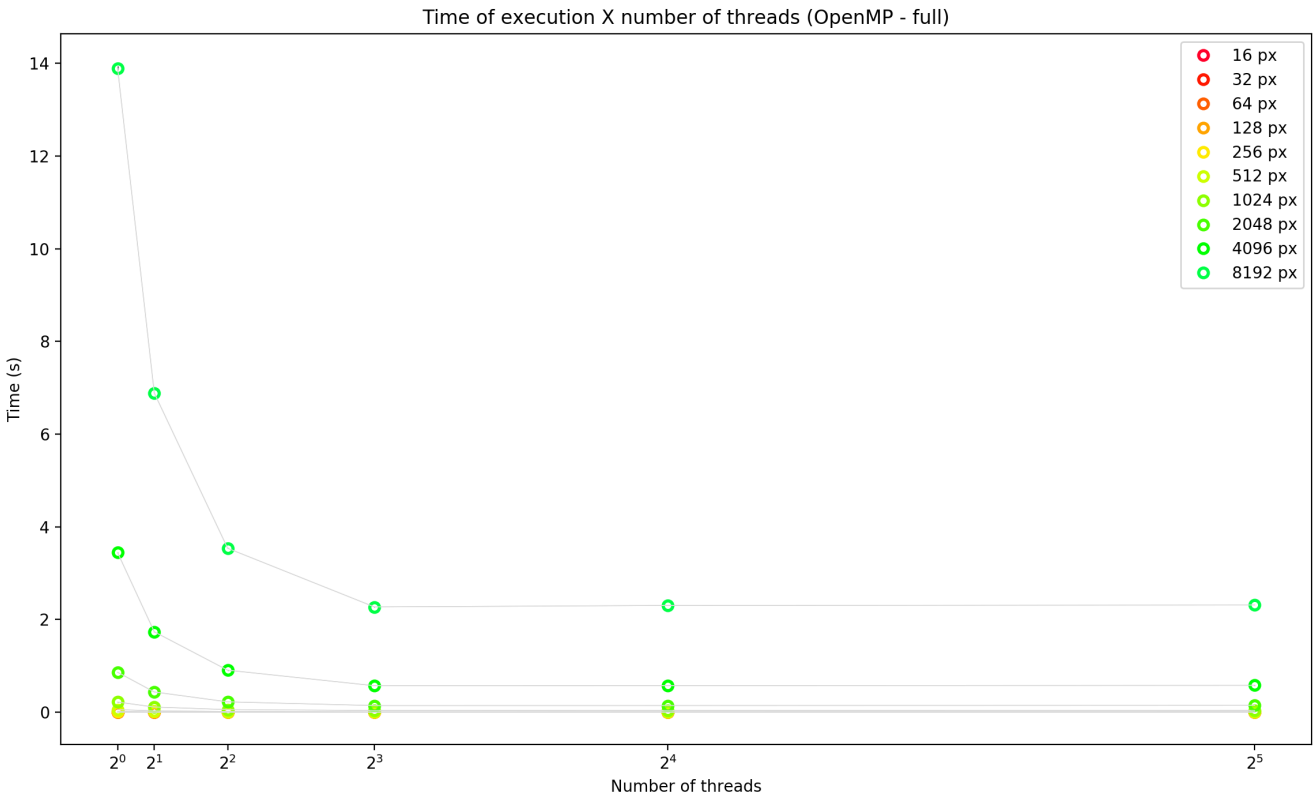
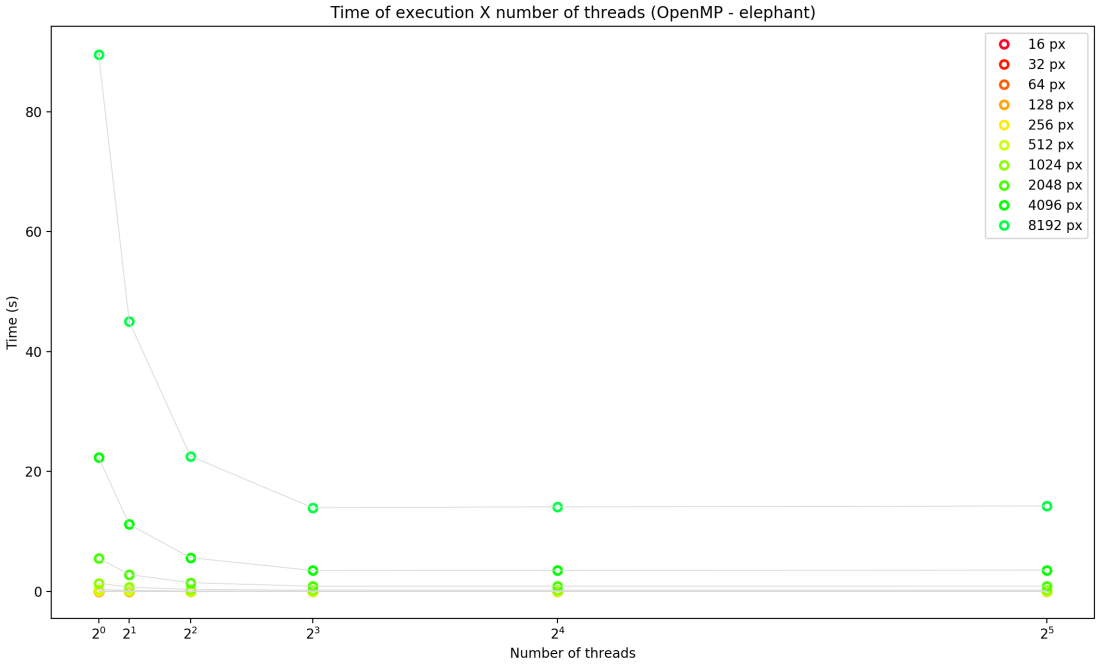
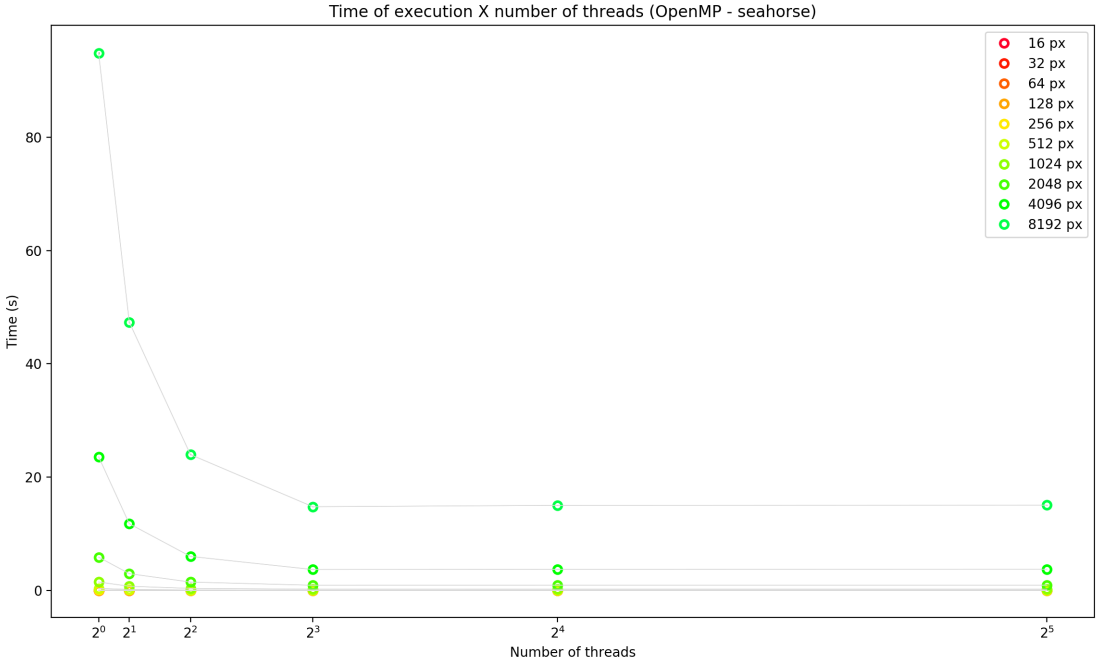
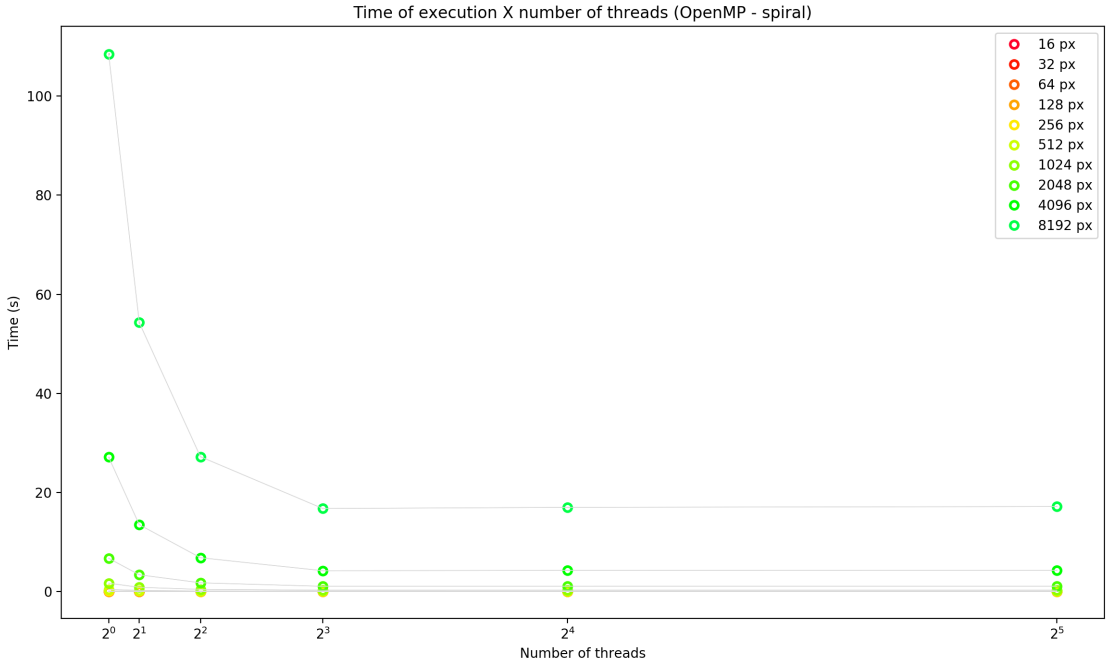


Figura 6: Esse gráfico mostra o tempo de execução X numero de threads para cada tamanho de entrada. Esses valores foram experimentados na região "full". Barras de erro também estão plotadas, porém são tão pequenas que talvez não estejam visíveis.

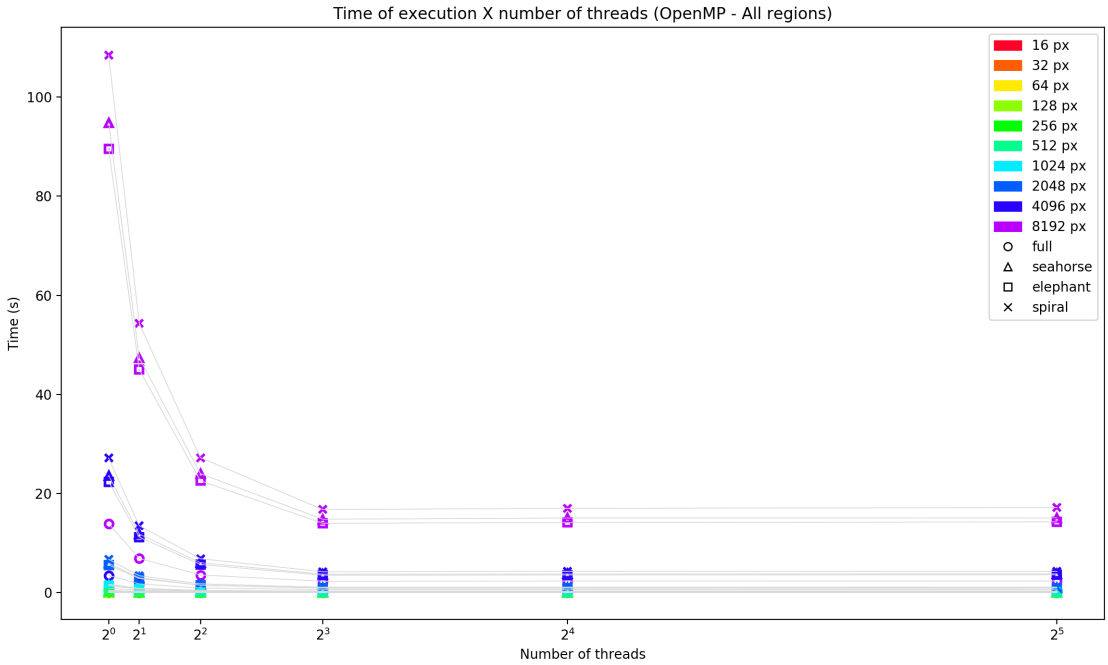
Percebemos que o aumento de threads de fato melhorou o tempo de execução do programa. No entanto ao ultrapassar 8, o mesmo número de cores da maquina testada, não houve grande melhora no tempo de execução. Esse mesmo comportamento foi observado para as demais regiões:


Time of execution X number of threads (OpenMP - elephant)
16 px
32 px
64 px
128 px
256 px
512 px
1024 px
2048 px
4096 px
8192 px
Time (s)
0
20
40
60
80
2^0
2^1
2^2
2^3
2^4
2^5
Number of threads


Time of execution X number of threads (OpenMP - seahorse)
16 px
32 px
64 px
128 px
256 px
512 px
1024 px
2048 px
4096 px
8192 px
Time (s)
0
20
40
60
80
2^0
2^1
2^2
2^3
2^4
2^5
Number of threads

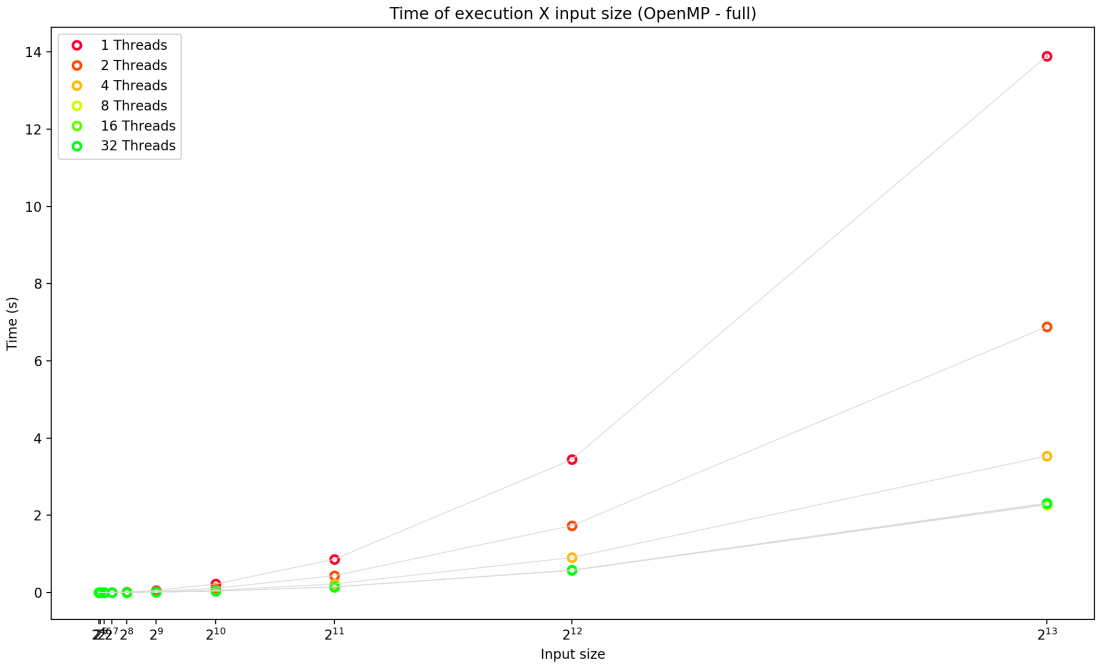


Nota-se que nessas regiões apesar do comportamento do tempo X numero de threads manter-se o mesmo, o tempo de execução aumenta, pois nessas regiões há mais cálculos de pontos com mais interações. Podemos comparar todas as regiões:



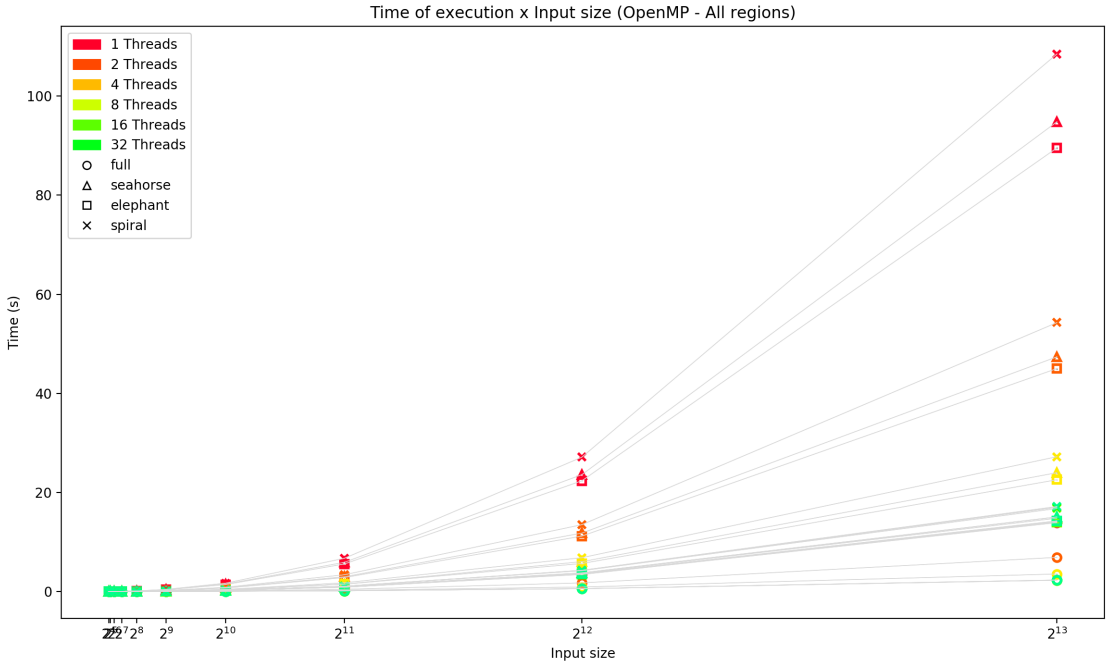
Vemos que as regiões Spiral, Seahorse e Elephant tem tempo de execução bastante maior que a região Full.

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:

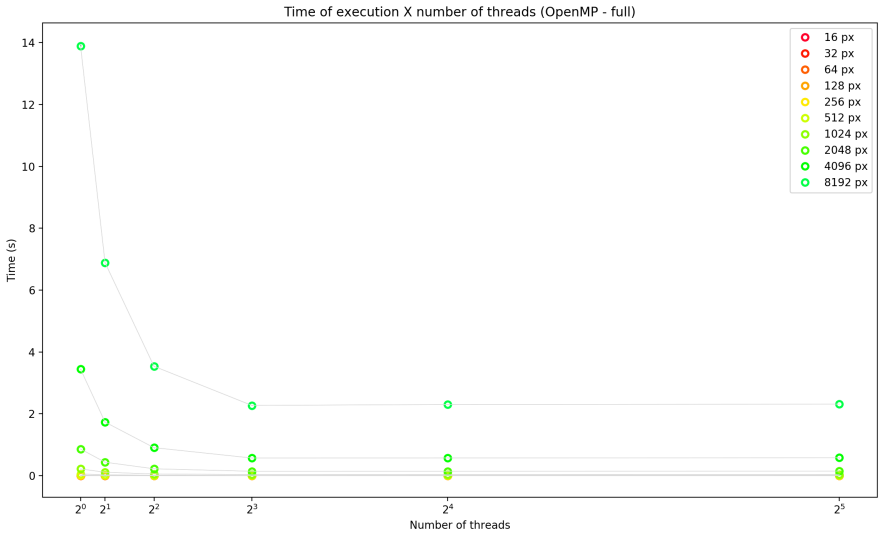


Vemos que há um aumento exponencial do tempo pelo tamanho da imagem. Vale lembrar que consideramos o tamanho de entrada, somente o tamanho do lado da imagem, portanto o comportamento exponencial se justifica por o número de pixels (chamadas para a função `compute_manderbolt()`) ser o tamanho do lado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

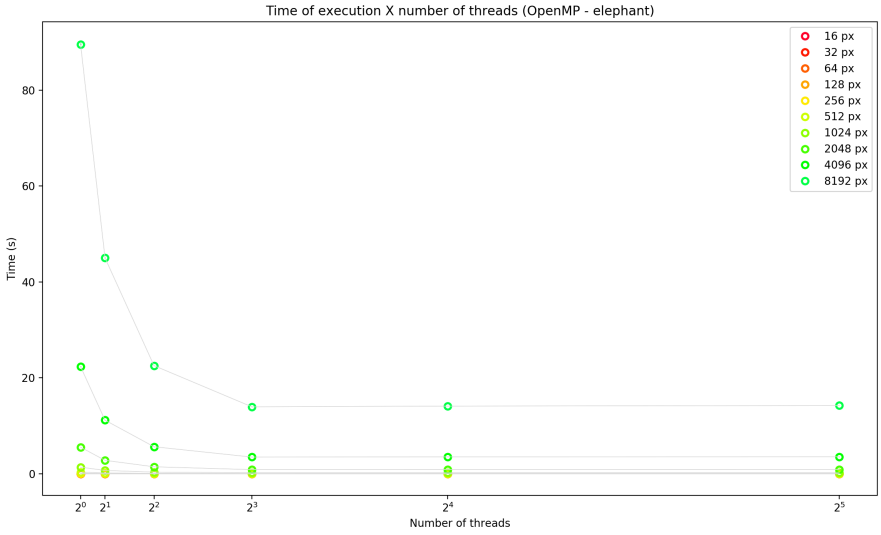
Nas demais regiões o comportamento é semelhante:



Outra API usada na paralelização é o OpenMP. Pela natureza do cálculo do conjunto Mandelbrot, como cada iteração possui tempo de execução diferente para cada pixel, utilizamos *dynamic scheduling*, sendo ela que oferece melhor distribuição de trabalho.



(a)



(b)

Figura 7: (b) representa uma ampliação dos dados de imagens de 2048 pixels da figura (a) .

Há um aumento de tempo de execução no cálculo do conjunto de regiões ampliadas como *spiral*, *elephant* devido ao maior número de cálculos envolvidos em uma região ampliada do nosso conjunto.

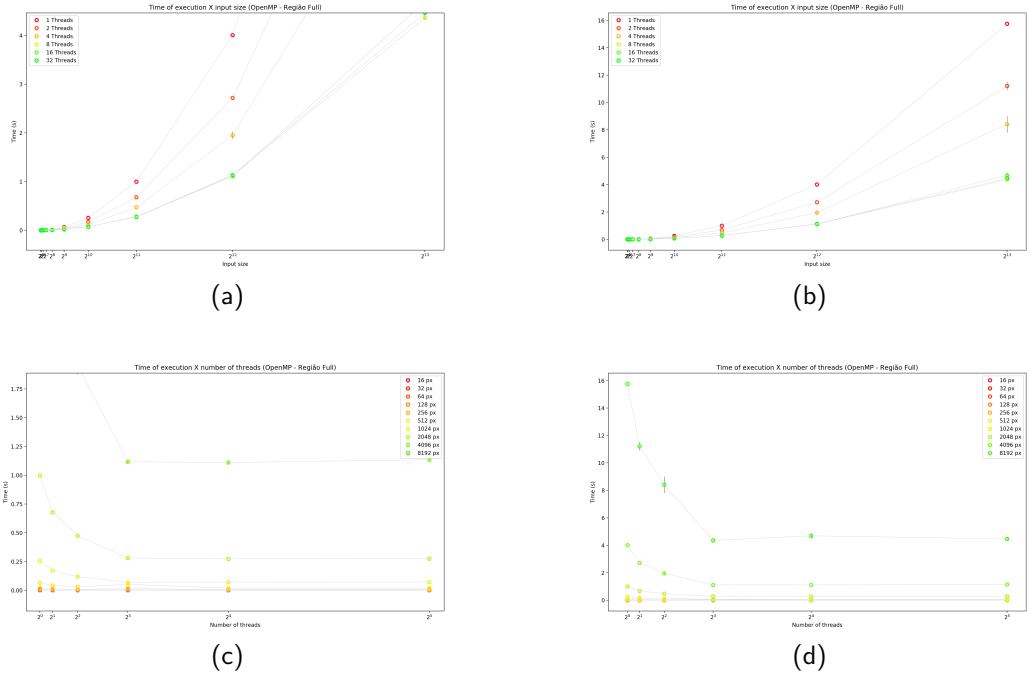


Figura 8: Temos a comparação de execução entre as 4 regiões do conjunto.

5 Conclusão