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, 1 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

O cálculo do conjunto em código sequencial ocorre sem nenhuma paralelização. No trabalho, a versão com alocação de memória e operações de IO chama-se mandelbrot_seq.c, enquanto que a versão sem a alocação e sem IO chama-se mandelbrot_seq_s.c. Definimos o tamanho do nosso image_buffer[11500 x 11500 x 3][3] como tamanho máximo para os nossos testes com imagens 11500px por 11500px.

A seguir, temos a comparação de performance entre uma versão com alocação e IO, com outra sem alocação e sem IO:

(insert image)

Possuindo uma matriz com o tamanho pré definido libera a necessidade de acessar a memória e fazer a alocação, dando nos resultados mais rápidos.

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 distribuido para cada thread.

3.1 Implementação com Divisão Estática

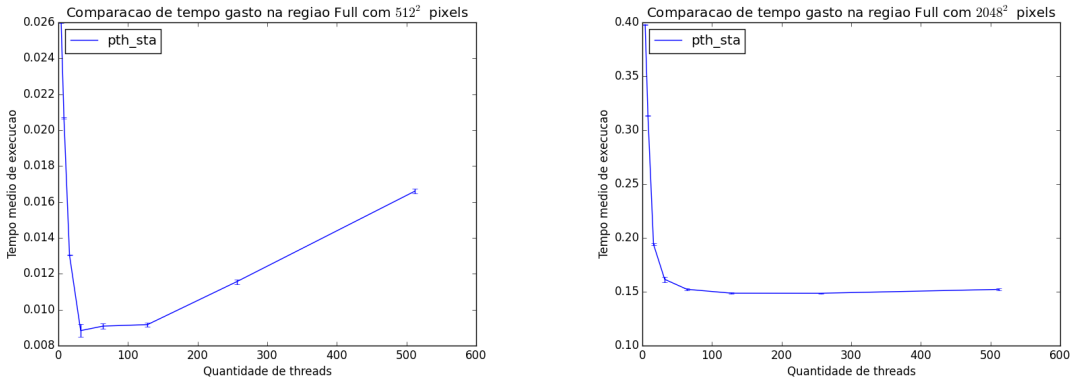
Chamamos de implementação com divisão estática a versão do nosso código em pthreads no qual o trabalho é dividido em n pedaços de mesmo tamanho, e cada pedaço é dado a uma thread. Chamamos essa implementação de estática porque cada thread recebe apenas um bloco de trabalho, pré-determinado pela divisão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

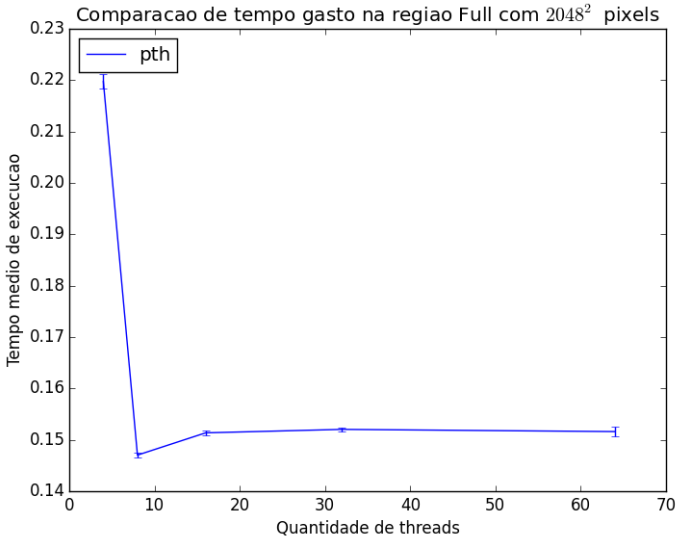
10: **end for**

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.

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 grandes o tempo gasto deve aumentar, porque caímos novamente no problema da implementação static, onde o trabalho não era bem dividido.



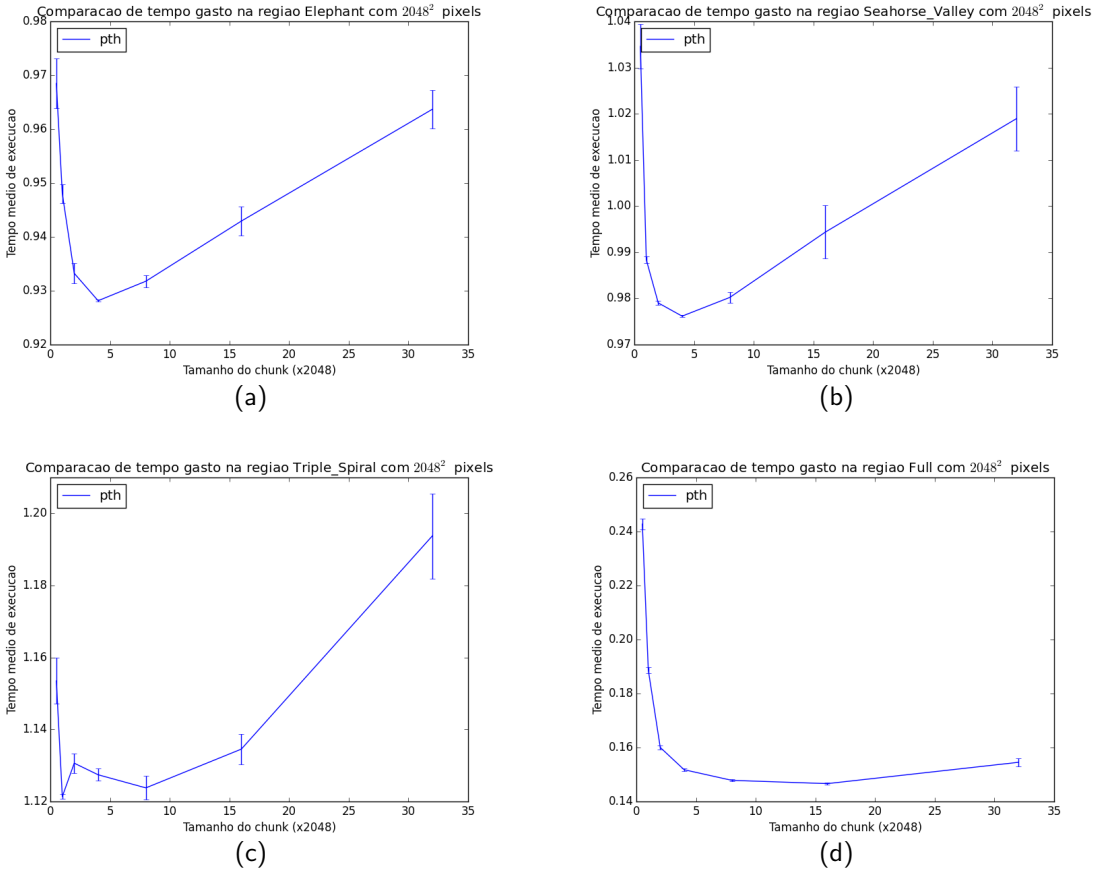
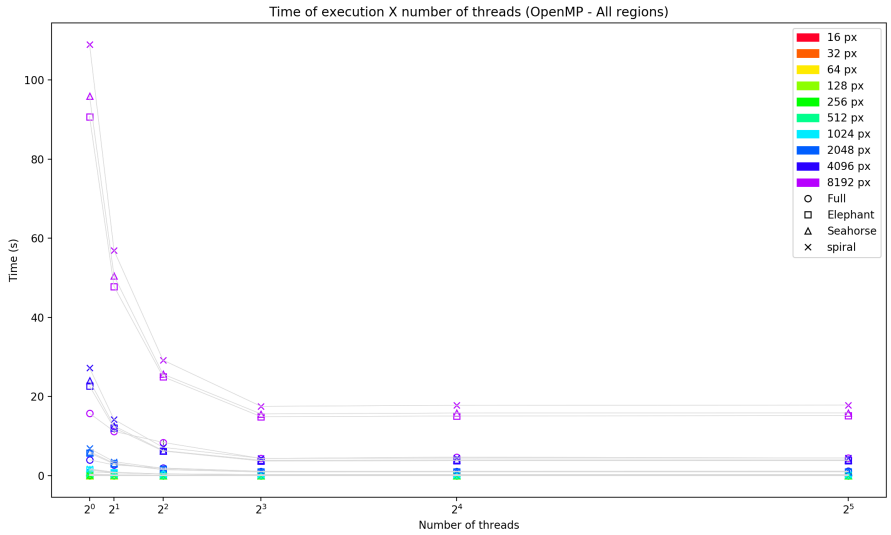


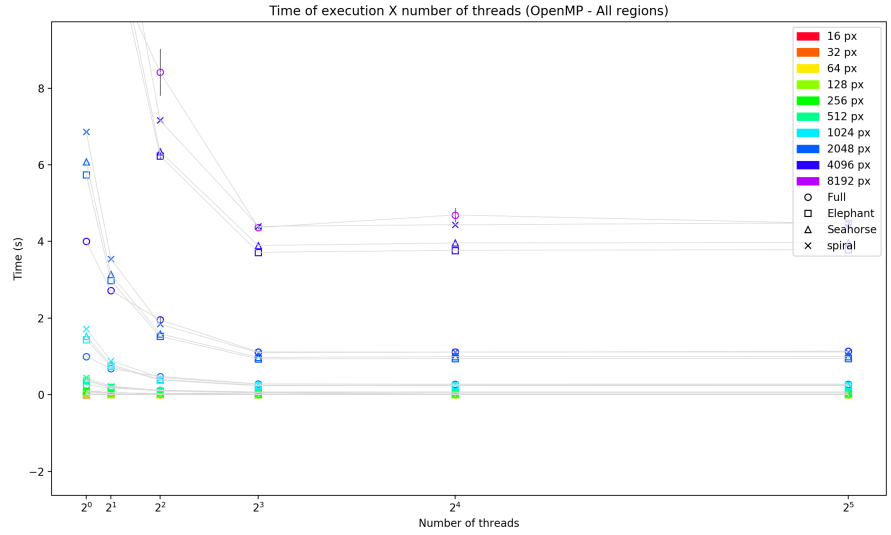
Figura 2: Verificamos empiricamente que definir o tamanho do chunk como o tamanho do lado da imagem traz bons resultados para o tempo de execução.

4 Código em OpenMP

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.



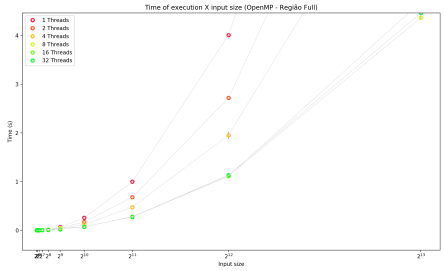
(a)



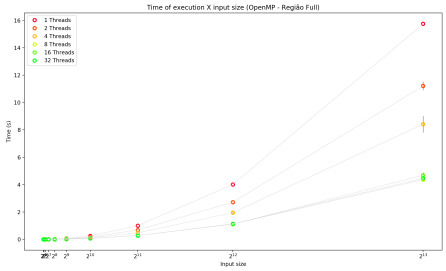
(b)

Figura 3: (b) representa uma ampliação dos dados de imagens de 2048 pixels da figura (a) .

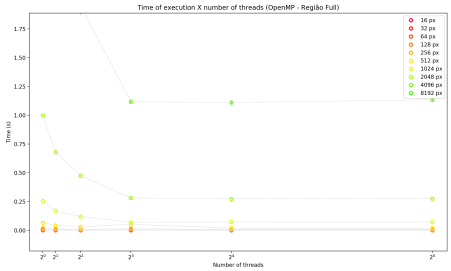
Há um aumento de tempo de execução no cálculo do conjunto de regiões ampliadas como *spiral*, *elephant* devido ao maior número de cálculos envolvidos em uma região ampliada do nosso conjunto.



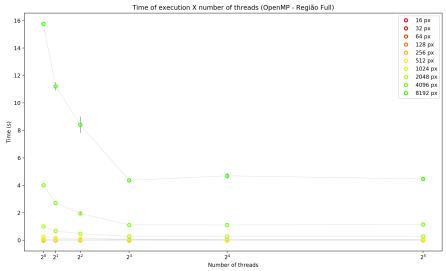
(a)



(b)



(c)



(d)

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

5 Conclusão