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, 24 de Abril de 2017

Conteúdo

1 Introdução **2**

2 Código Sequencial **2**

3 Código em OpenMP **2**

4 Código em Pthreads **2**

4.1 Implementação com Divisão Estática 2

4.2 Implementação com Divisão Dinâmica 3

5 Discussões Gerais **4**

6 Conclusão **4**

1 Introdução

2 Código Sequencial

3 Código em OpenMP

4 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.

4.1 Implementação com Divisão Estática

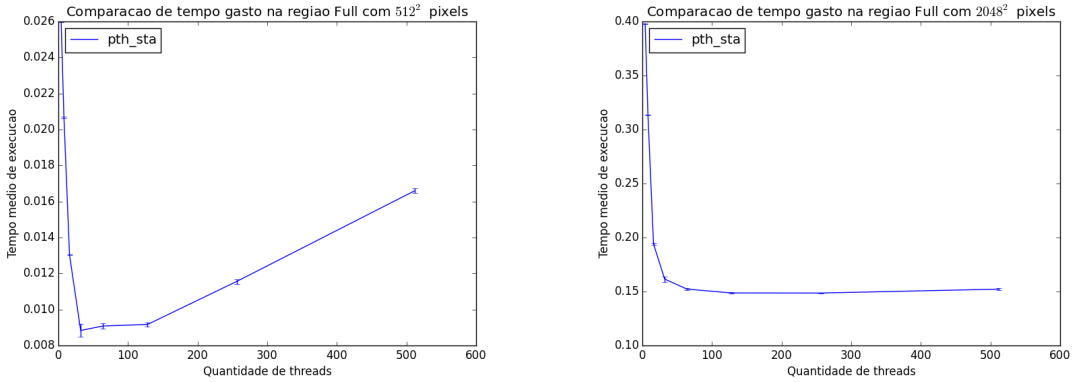
Chamamos de implementação com divisão estática a versão do nosso código em pthreads no qual o trabalho é dividido em n pedaços de mesmo tamanho, e cada pedaço é dado a uma thread. Chamamos essa implementação de estática porque cada thread recebe apenas um bloco de trabalho, pré-determinado pela divisão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

4.2 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, dinâmicamente, delegar blocos de pixels a cada thread. Portanto, agora diminuimos 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 ← 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.

Diminuido 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.

5 Discussões Gerais

6 Conclusão