Instituto de Matemática e Estatística

EP3 - MAC0219

Cálculo do Conjunto de Mandelbrot em Paralelo com OpenMPI

Professor: Alfredo Goldman

Alunos: Bruno Sesso

Gustavo Estrela de Matos

São Paulo, 9 de Julho de 2017

Conteúdo

1 Introdução

Neste trabalho, implementaremos uma versão paralela, em OpenMPI, do código que calcula o fractal de uma região de Mandelbrot. Esta versão deve ser capaz de, dado um cluster de computadores, utilizar o padrão de troca de mensagens MPI para distribuir e reunir em um nó principal os cálculos necessários para gerar a imagem da região pedida.

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

O código sequencial foi fornecido no enunciado do EP e com ele fizemos medições de tempo para diferentes regiões do plano complexo.

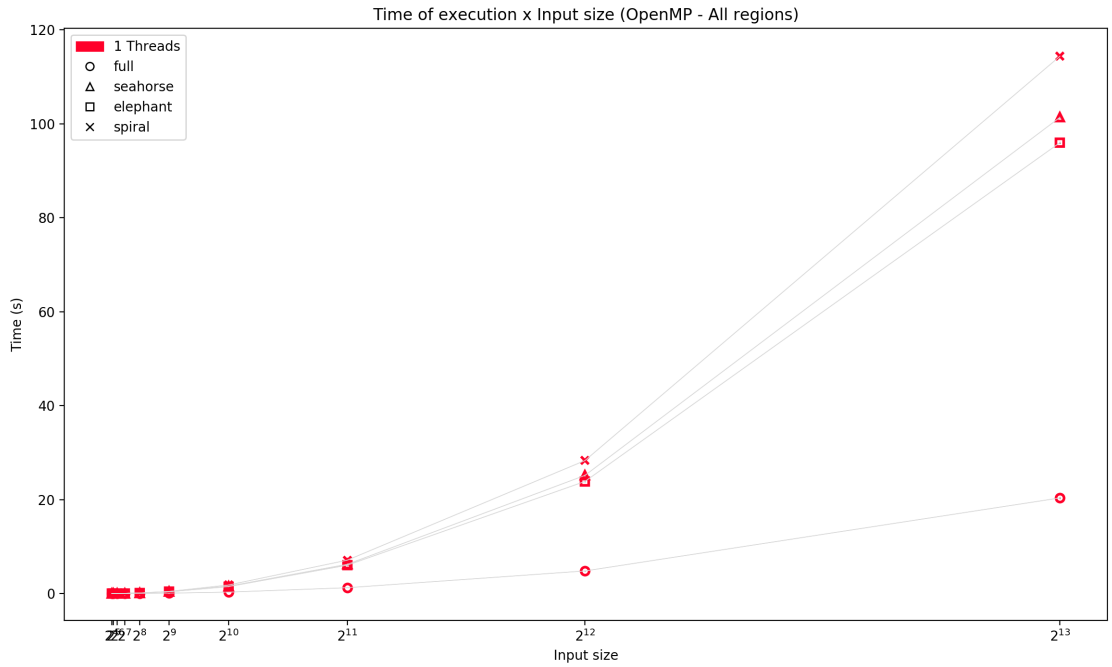

Figura 1: Tempo de execução do programa sequencial para diferentes tamanhos de entrada

Nota-se que o tempo em cada região é diferente, pois em cada região pode haver mais cálculos de pontos com mais interações. Em especial, podemos observar que a região Full é a mais fácil de ser calculada.

Como explicamos no EP1, o cálculo dessas regiões escala de uma maneira, aparentemente, mais do que linear. Além disso, o consumo de tempo causado por operações de alocação, leitura e escrita de memória são lineares e serão desconsideradas nesse trabalho.

3 Código em OpenMPI

A nossa implementação em OpenMPI define uma máquina como principal (mestre), e esta deve ser capaz de dividir o trabalho entre as outras máquinas (escrava), assim como reunir os resultados para construir a imagem da região Mandelbrot. Para fazer isso, dividimos os pontos da imagem em blocos de pixels e, então a máquina principal envia para outras máquinas os blocos que devem ser calculados, por meio de uma mensagem que indica o índice do bloco que deve ser calculado. Depois de calcular o bloco, a escrava envia para mestre um vetor que contém as iterações calculadas naquele bloco para compor a imagem.

A estrutura de blocos que usamos neste trabalho é a mesma usada no EP1 e, como fizemos na última vez, vamos dividir a imagem em n blocos, em que n é o tamanho do lado da imagem, isto é, cada bloco corresponde a uma linha da imagem. Além disso, como constatamos no EP1 que uma divisão dinâmica de trabalho é mais vantajosa, vamos adotar essa abordagem também neste trabalho. Ao invés de pré-determinar em qual máquina um bloco será calculado, delegamos a máquinas livres blocos que ainda não foram calculados.

Para implementar essa divisão dinâmica de trabalho, a máquina mestre deve iniciar o processo delegando um bloco para cada máquina escrava. Depois, sempre que uma máquina enviar o trabalho pronto, a mestre deve processar essa resposta, atualizando o buffer da imagem, e enviar a escrava um novo bloco para ser calculado (se houver). Para finalizar o processo, a mestre também deve enviar um sinal de "trabalho feito"para todas escravas, o que faz possível terminar o programa em todas as máquinas do cluster.

Veja abaixo um pseudo-código da função que roda na máquina mestre, distribuindo os blocos de pixels para as máquinas escravas:

1: **function** DistributeWork
2: $S \leftarrow$ lista de nacos de todos os pixels
3: $sent\_chunks \leftarrow 0$
4: $received\_chunks \leftarrow 0$
5: **for all** M máquina escrava do cluster **do**
6: $Send(M, S[sent\_chunks])$
7: $sent\_chunks \leftarrow sent\_chunks + 1$

```
8:      end for
9:      while received_chunks < |S| do
10:         N ← PullJobFromAnyMachine()          ▷ Recebe trabalho de escrava N
11:         received_chunks ← received_chunks + 1
12:         if sent_chunks < |S| then
13:             Send(N, S[sent_chunks])
14:             sent_chunks ← sent_chunks + 1
15:         else
16:             Send(N, END_CODE)
17:         end if
18:     end while
19: end function
```

4 Conclusão