Trabalho final de Computação Paralela
Mandelbrot

Daniel Dias de Lima[1], **Pedro Henrique Nascimento Santos**[1]

[1]Faculdade de Computação e Informática – Universidade Presbiteriana Mackenzie (UPM)
São Pauulo – SP – Brasil

danieldias.lima@mackenzista.com.br, thunder.pedro@hotmail.com

Resumo. *O presente trabalho consiste na exploração de otimizações na realização de cálculos para a obtenção do fractal de Mandelbrot. Para tal, foi realizado o uso de recursos computacionais que possibilitam a paralelização das diversas tarefas que compoem o problema. O presente trabalho não se propõe em contextualizar o problema, mas em entender quais são as diversas tarefas que compoem sua solução e as diversas medidas usadas em suas otimizações, e os resultados obtidos por tais medidas.*

1. Descrição inicial do problema

Como ponto de partida para a paralelização do problema apresentado, foi feito um estudo sobre as tarefas que compoem o problema e suas dependências. Como pode ser observado através do gráfico de dependências representado na Figura 1, o problema do fractal de Mandelbrot é de natureza facilmente paralelizável, uma vez que a maior parte das tarefas o compoem não possuem dependência entre si, fazendo com que boa parte do trabalho desenvolvido tenha se focado na otimização da forma como as tarefas são distribuídas entre as unidades de processamento disponíveis, assim como veremos em na seção 3.

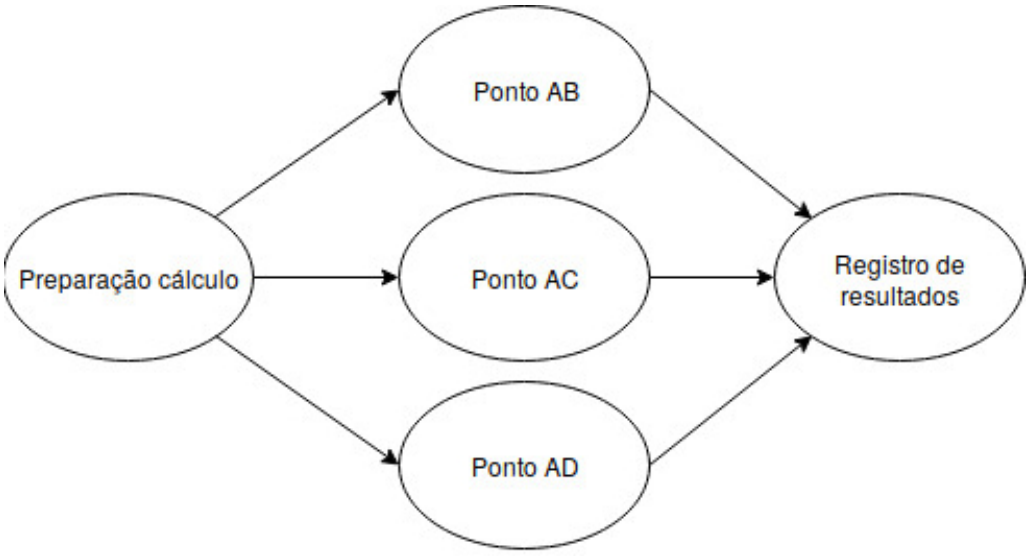

Figura 1. Gráfico de dependência de tarefas

Outras considerações que precisam ser apontadas são as tarefas apresentadas no começo e no final do nosso gráfico de dependências. Chamamos de Preparação Cálculos,

todas as tarefas realizadas previamente ao cálculo dos pontos da matriz, e.g. o recebimento das entradas e a alocação de memória necessária. Como a realização de tais atividades não apresentou um tempo mensuravelmente significante em nossa análise, decidimos desviarmos nossos esforços das mesmas.

E o mesmo foi feito em relação a atividade de Registro de Resultados, que embora pudesse ser considerável para entradas que resultassem em matrizes muito grandes, também não se apresentou como uma atividade facilmente otimizável, considerando suas dependências, e a natureza de I/O (Input/Output) que a mesma apresenta.

2. Metodologia de trabalho

Para que pudessemos ter um melhor entendimento sobre as possíveis soluções do problema, partimos do programa base que nos foi dado, e fizemos o uso do OpenMP [OpenMP 2013] para a paralelização dos *nested loops* usados para a iteração sobre a matrix usada para o registro dos resultados.

Considerando a natureza de independência dos cálculos de cada um dos pontos, poderíamos ter os pares de variáveis c e r executados em qualquer ordem, contanto que todos os pares fossem considerados. A solução a seguir faz exatamente isso.

```
#pragma omp parallel
{
     #pragma omp for collapse(2) schedule(static)
     for (int r = 0; r < max_row; ++r) {
          for (int c = 0; c < max_column; ++c) {
               //realiza  o do calculo do ponto (r,c)
        }
    }
}
```

Listing 1. Iteração sobre os valores x,y da matriz resultado

Em um primeiro momento, ainda realizávamos o cálculo do valor a ser somado com o número complexo de interesse, o z, dentro da repetição *while*, laço esse que ocorre uma quantidade indeterminada de vezes, até que o valor exceda 2 ou a quantidade de iterações passadas como parâmetro inicialmente seja quebrada.

```
complex<float> z, next;
int iterations = 0;
while (abs(z) < 2 && ++iterations < max_n) {
     next = decltype(z)((float)c * 2 / max_column - 1.5, (float)r * 2 /
          max_row - 1);
     z = pow(z, 2.0f) + next;
}
```

Listing 2. Iteração usado no cálculo do ponto x,y

Com a solução apresentada, conseguimos os resultados de speedup e eficiência como apresentado a seguir, nas figuras 2 e 3. Isso nos permite um questionamento inicial sobre o problema, para que nós possamos explorar outras melhorias e termos uma comparação base com as propostas que seguem. Os valores absolutos obtidos nesses testes se encontram na tabela 1.

Tabela 1. Resultados solução base

Threads	Speedup	Eficiência
2	1.876456876	0.938
4	2.471220261	0.617
8	4.30625209	0.538
16	8.177777778	0.511
32	15.02917153	0.469
64	26.77754678	0.418
128	44.56747405	0.348
256	61.92307692	0.241

Inicialmente consideramos que o *overhead*, explanado por [Barney et al. 2010], causado pela paralelização do problema usando o pragma *omp for*, que realiza a distribuição das tarefas usando o tipo de *schedule* estático, poderia ser o problema. Mas considerando a queda causada da troca de uma para duas *threads*, não seria justificado justamente pelo tipo de schedule de menor custo dentro da especificação OpenMP [Bull 1999].

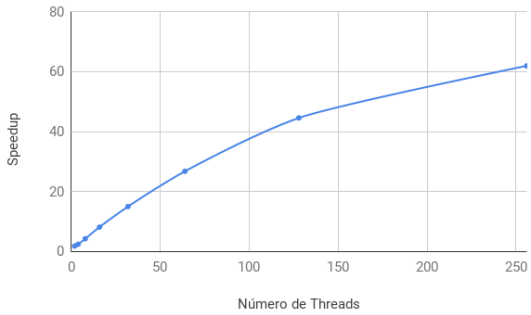

Figura 2. Speedup solução base

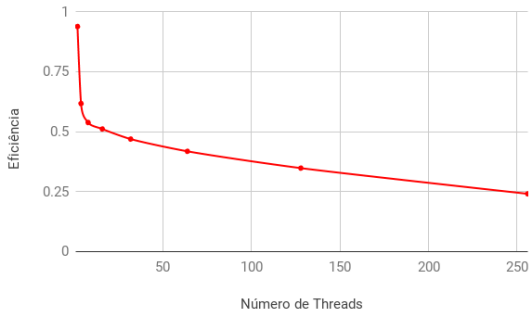

Figura 3. Eficiência solução base

2.1. Ponderações sobre speedup e eficiência da solução base

Não sabíamos o quão eficiente tínhamos sido com os resultados obtidos inicialmente, e por isso a pesquisa seguiu, dessa vez, porém, com uma base de comparação. Alguns questionamentos que conseguimos colocar de cara, porém, incluem uma queda brusca na eficiência da nossa solução base a partir do uso de uma segunda thread, e considerando os valores do *speedup*, o motivo do mesmo não ter crescido proporcionalmente com a quantidade de entradas, considerando a paralelização das tarefas, conforme descrito anteriormente.

3. Divisão do trabalho entre as unidades

Levando em conta as considerações levantadas anteriormente, pudemos então levantar questionamentos sobre quais seriam os melhores tipos de *schedule* que poderíamos usar dentro da nossa solução.

Considerando o tipo de trabalho realizado, chegamos a conclusão de que o tipo dinâmico de *schedule* seria mais apropriado. E isso se dá por causa das diferenças de

tamanhos das atividades realizadas para cada um dos pontos: os pontos que estão dentro do mandelbrot são muito mais custosos computacionalmente, porque por definição eles ultrapassam os números de iterações definidos na entrada, e fazendo uso do tipo estático de schedule, *chuncks* específicos de trabalho poderiam ser compostos exclusivamentes por esses pontos, o que traria uma carga de trabalho desigual entre as unidades de processamento.

E como havíamos apontado na seção 1, trouxemos o cálculo do próximo número a ser somado ao número complexo de interesse para fora do nosso loop de cálculo de ponto, uma vez que essa operação é mantida a mesma e não muda com as iterações.

O código da nossa solução final ficou conforme apresentado a seguir.

```
#pragma omp parallel
{
     #pragma omp for collapse(2) schedule(dynamic)
     for (int r = 0; r < max_row; ++r)
     {
          for (int c = 0; c < max_column; ++c)
          {
               complex<float> z, next;

               next = decltype(z)((float)c * 2 / max_column - 1.5, (
                   float)r * 2 / max_row - 1);

               int iterations = 0;
               while (abs(z) < 2 && ++iterations < max_n)
                    z = pow(z, 2.0f) + next;

               mat[r][c] = (iterations == max_n ? '#' : '.');

          }
     }
}
```

Listing 3. Código final

4. Resultados da solução final

Começamos a exposição dos resultados obtidos com o speedup e eficiência obtidos na execução do ambiente BOCA, assim como na seção 1: a tabela 2 apresenta os valores absolutos, e as figuras 4 e 5 os mesmos resultados de uma forma gráfica para uma melhor visualização.

4.1. Problemas levantados na solução base

Através das otimizações realizadas, podemos observar uma queda menos brusca no gráfico de eficiência na figura 5. O *overhead* causado pela paralelização seria inevitável, especialmente considerando a quantidade de iterações e o uso do *schedule dynamic*.

Localmente, os testes foram realizados em uma unidade de processamento com 4 *threads* físicas e 4 *threads* lógicas, e os resultados de speedup e eficiência podem ser observados nas figuras 6 e 7, respectivamente. Os valores brutos de tais resultados se encontram na tabela 3.

Tabela 2. Resultados solução otimizada (boca)

Threads	Speedup	Eficiência
2	2.067747632	1.033
4	3.901847925	0.975
8	7.820279296	0.977
16	14.5045045	0.906
32	26.94560669	0.842
64	45.03496503	0.703
128	65.38071066	0.51
256	79.01840491	0.308

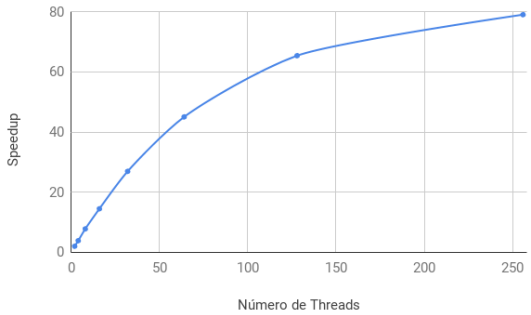

Figura 4. Speedup BOCA

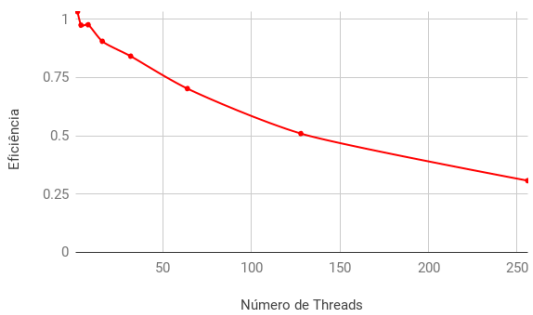

Figura 5. Eficiência BOCA

Os valores de entrada para a realização de tais testes foram 300 para a quantidade de linhas, 1000 para as colunas e 1000 também para a quantidade de iterações máximas.

4.2. Escalabilidade fraca

Assim como descrito por [Li 2019], a escalabilidade fraca está diretamente relacionada com a lei de Gustafson, e diz respeito ao speedup que é calculado com base em um tamanho de problema que cresce proporcionalmente aos recursos disponíveis para o processamento (no nosso caso, a quantidade de *threads*).

Os testes de escalabilidade fraca foram feitos apenas localmente, e o crescimento do tamanho do nosso problema se deu na quantidade de pontos calculados; mais especificamente, a quantidade de pontos no eixo x.

As quantidades fornecidas de entrada, e os resultados obtidos nos nossos testes podem ser observados na figua 8.

Tabela 3. Resultados solução otimizada (local)

Threads	Tempo bruto	Speedup	Eficiência
1	40.78	1	1
2	21.38	1.907390084	0.953
4	12.07	3.378624689	0.844
8	9.12	4.471491228	0.558

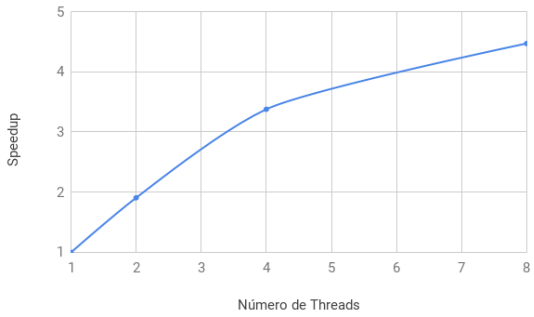

Figura 6. Speedup local

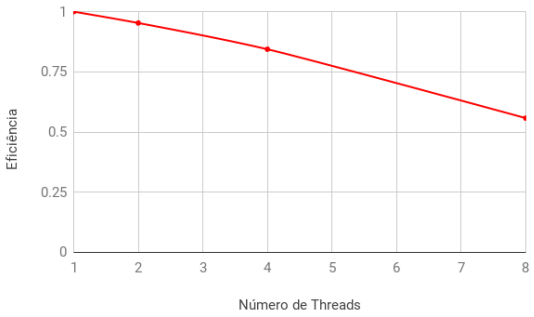

Figura 7. Eficiência local

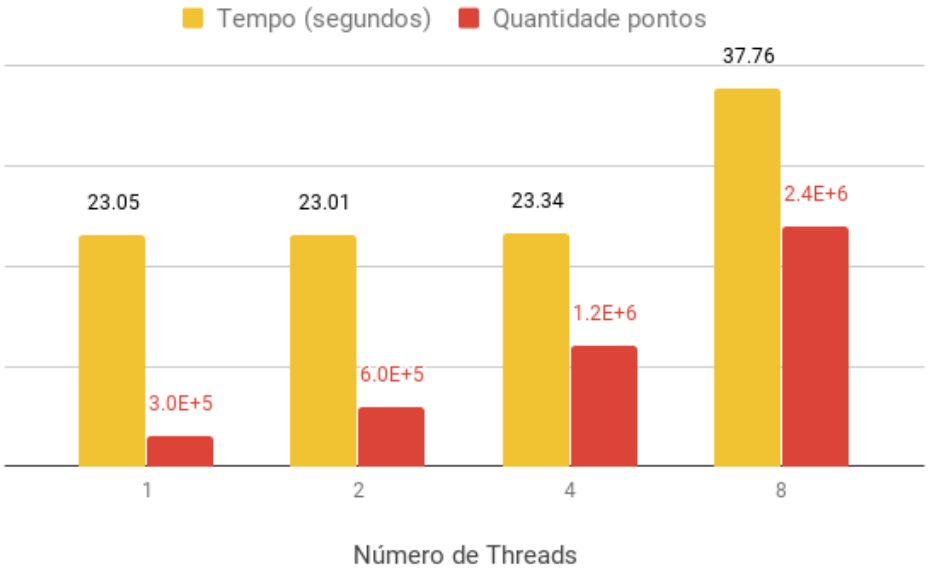

Figura 8. Resultados dos testes de escalabilidade fraca

Como os resultados demonstram, e assim como era esperado, o tempo de prossamento para cada uma das entradas se manteve mais ou o mesmo até 4 *threads*. Contudo, o uso de 8 *threads* apresenta um resultado diferente do padrão obtido até então, e isso provavelmente foi causado pela transição entre *threads* físicas e *threads* lógicas.

5. Possíveis melhoras

Uma possível melhora a ser feita em cima do presente trabalho seria a forma como nós calculamos cada um dos pontos. O conceito de pertubação, que usa um ponto base como referência para os cálculos dos demais pode ser usado para melhor eficiência no cálculo de iterações com maiores ”zooms”, em intervalos mais delimitados.

Referências

Barney, B. et al. (2010). Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10.

Bull, J. M. (1999). Measuring synchronisation and scheduling overheads in openmp. In *Proceedings of First European Workshop on OpenMP*, volume 8, page 49.

Li, X. (2019). Scalability: strong and weak scaling.

OpenMP, A. (2013). Openmp application program interface version 4.0.