CÁLCULO ESTIMATIVO DA ÁREA DE UM CONJUNTO DE MANDELBROT USANDO MPI E OPENMP[1]

Marcelo Bernardy de Azevedo[2] <marcelo.bernardy@acad.pucrs.br>
Thomas Pozzer Moraes[3] <thomas.pozzer@acad.pucrs.br>
Roland Teodorowitsch[4] <roland.teodorowitsch@pucrs.br> – Orientador

Pontifícia Universidade Católica do Rio Grande do Sul – Faculdade de Informática – Curso de Engenharia de Software
Av. Ipiranga, 6681 Prédio 32 Sala 505 – Bairro Partenon – CEP 90619-900 – Porto Alegre – RS

25 de Junho de 2020

RESUMO

Este artigo descreve uma solução para o segundo trabalho proposto na disciplina de Programação Paralela do curso de Engenharia de Software na PUCRS, que aborda a paralelização da solução da Área de um Conjunto de Mandelbrot usando MPI e OpenMP, incluindo a avaliação de desempenho do algoritmo e contextualização sobre a utilização do processamento paralelo para obtenção de um melhor desempenho.

Palavras-chave: MPI; OpenMP; Mandelbrot; Programação Paralela;

ABSTRACT

Title: *"Area of a Mandelbrot set using MPI and OpenMP"*

This article describes a solution for the second work proposed in the Parallel Programming discipline of the Software Engineering course at PUCRS, which approach the parallelization of the Mandelbrot Set Area solution using MPI and OpenMP, including the performance evaluation of the algorithm and contextualization on the use of parallel processing to obtain better performance.

Key-words: *MPI; OpenMP; Mandelbrot; Parallel Programming;*

1 INTRODUÇÃO

O problema descrito na proposta do trabalho é de calcular a área de um conjunto de Mandelbrot. O Conjunto de Mandelbrot é um conjunto de números complexos c para os quais, a partir de uma condição inicial z=c, a iteração de $z=z^2+c$ não diverge. Para determinar se um ponto c está neste conjunto, deve-se realizar determinado número de iterações sobre z, sendo que, se ao final dessas iterações tivermos $|z| > 2$, então o ponto está fora do conjunto de Mandelbrot.

Este artigo mostrará a experiência de paralelizar uma versão sequencial de um código que realiza um cálculo estimativo da Área de um Conjunto de Mandelbrot (BULL, 2019). Nesse código, geram-se pontos de forma homogênea sobre uma área quadrada e para cada ponto realiza-se a iteração desse ponto usando a fórmula indicada 2000 vezes. Se depois dessas 2000 iterações o módulo de z for maior do que 2, então o ponto é considerado fora do Conjunto de Mandelbrot. Contando-se quantos dos pontos estão dentro do conjunto, pode-se obter uma estimativa da área do conjunto.

Para obter a real experiência de paralelização foi utilizado os *Clusters* cedido pela universidade realizando testes com 2, 3 e 4 *nodos*, além do sequencial que utiliza apenas uma *thread*. Para a paralelização do conjunto de Mandelbrot é empregada a utilização da Interfaces MPI e OpenMP.

2 ANÁLISE DA PARALELIZAÇÃO

Para a identificação da possibilidade de paralelização o primeiro passo foi entender o código de forma sequencial disponível pelo professor. Após foi realizado o desenvolvimento do algoritmo utilizando MPI e outro unindo MPI com OpenMP.

1 Artigo elaborado como segundo trabalho da disciplina da de Programação Paralela.

2 Aluno de graduação da disciplina de Programação Paralela do curso de Engenharia de Software, da Faculdade de Informática da PUCRS.

3 Aluno de graduação da disciplina de Programação Paralela do curso de Engenharia de Software, da Faculdade de Informática da PUCRS.

4 Professor das disciplinas de Introdução à Ciência da Computação e Programação Distribuída do curso de Ciência da Computação, e da disciplina de Programação Paralela à Engenharia de Software, da Faculdade de Informática da PUCRS.

2.1 Análise do código

Após a análise do código, pode-se concluir que dos 3 laços existentes no código sequencial, apenas um pode ser paralelizado. O laço mais interno não pode ser paralelizado pois há o comando *break*, já o laço intermediário causaria um número desnecessário de criação de *threads* pois a cada iteração do laço mais externo teria a necessidade de recriar as *threads*, dito isso, o laço escolhido a ser paralelizado é o mais externo pois não apresenta nenhuma restrição para o seu paralelismo.

Cabe ressaltar o tratamento da seção crítica presente dentro do laço mais interno na variável *value*, para com que o valor seja incrementado de forma correta, além das variáveis privadas para cada *thread c, z* e *ztemp, numoutside.*

2.2 *MPI*

Segundo Gropp et al. (1994), Foster (1995) e Pacheco (1999), o MPI (Message Passing Interface) é um padrão de interface para a troca de mensagens em máquinas paralelas com memória distribuída. Por isso o padrão MPI é referido como MPMD (*Multiple program multiple data*).

A utilização do MPI se dá através de algumas rotinas dentre estas pode-se destacar:

- **MPI_INIT**: Inicializa o ambiente de execução de MPI.

- **MPI_Comm_size:** Determina o número de processos em um grupo associado a um comunicador.

- **MPI_Comm_rank:** Utilizado para associar um *rank* do processo requisitante ao comunicador.

- **MPI_Abort:** Termina todos os processos MPI associados com o comunicador.

- **MPI_Get_processor_name:** Retorna o nome do processador. Também retorna o tamanho do nome.

- **MPI_Initialized:** Usado para saber se MPI_Init já foi chamado.

- **MPI_Wtime:** Retorna o tempo decorrido em segundos no processador requisitante.

- **MPI_Finalize:** Termina a execução do ambiente MPI.

2.3 OpenMP

Para a paralelização do código sequencial foi utilizada a *API* (application programming interface) *OpenMP* (Open Multi-Processing). Ela trabalha com o modelo de execução *fork-join* e dentre suas vantagens é permitir a paralelização do programa com pequenas modificações no código-fonte.

A utilização do *OpenMP* se dá através de diretivas de compilação iniciadas com *#pragma omp alguns exemplos abaixo:*

- ***parallel***: precede um bloco de código que será executado em paralelo

- ***for***: precede um laço, for, com iterações independentes que pode ser executado em paralelo

- ***parallel*** for : uma combinação das primitivas, *parallel*, e, *for*

- ***critical*** : precede uma seção crítica do código

3 IMPLEMENTAÇÃO

3.1 MPI

Abaixo a implementação do código paralelo utilizando as rotinas do MPI em conjunto com a análise realizada na seção anterior.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#define MAXITER 2000

struct complex
{
    double real;
    double imag;
};

int main(int argc, char *argv[])
{
    int rank;
    int size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    for (int NPOINTS = 500; NPOINTS <= 5000; NPOINTS += 500)
    {
        int i, j, iter, numoutside, value = 0;
        double area, error, ztemp;
        double start, finish;
        struct complex z, c;

        start = MPI_Wtime();

        for (i = rank; i < NPOINTS; i += size)
        {
            for (j = 0; j < NPOINTS; j++)
            {
                c.real = -2.0 + 2.5 * (double)(i) / (double)(NPOINTS) + 1.0e-7;
                c.imag = 1.125 * (double)(j) / (double)(NPOINTS) + 1.0e-7;
                z = c;
                for (iter = 0; iter < MAXITER; iter++)
                {
                    ztemp = (z.real * z.real) - (z.imag * z.imag) + c.real;
                    z.imag = z.real * z.imag * 2 + c.imag;
                    z.real = ztemp;
                    if ((z.real * z.real + z.imag * z.imag) > 4.0e0)
                    {
                        value++;
                        break;
                    }
                }
            }
        }

        MPI_Reduce(&value, &numoutside, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        finish = MPI_Wtime();

        if (rank == 0)
        {
            area = 2.0 * 2.5 * 1.125 * (double)(NPOINTS * NPOINTS - numoutside) / (double)(NPOINTS
* NPOINTS);
            error = area / (double)NPOINTS;

            printf("NPOINTS: %d | Area of Mandlebrot set = %12.8f +/- %12.8f\n", NPOINTS, area,
error);
            printf("Time = %12.8f seconds\n", finish - start);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Figura 2 – Código fonte da versão paralela em MPI (programa em C)

Com essa implementação conseguimos alcançar os mesmos resultados do que com a versão sequencial no entanto com um aumento significativo de performance. Na próxima seção será explicada de uma forma mais detalhada o ganho de performance.

Pode-se observar acima a adição das variáveis *rank*, para associar o *rank* ao processo que o comunicador requisita, *size*, para determinar o número de processos. Também pode-se destacar o uso da rotina *MPI_Reduce* função usada para realizar a soma de todos os valores entre os processos.

3.2 MPI e OpenMP

Segue abaixo a implementação do padrão MPI em conjunto com OpenMP que obtiveram resultados bem mais rápidos quando comparados com a versão sequencial e também melhores quando comparados com a versão paralela utilizando apenas MPI.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <omp.h>

#define MAXITER 2000

struct complex
{
    double real;
    double imag;
};

int main(int argc, char *argv[])
{
    int rank;
    int size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    for (int NPOINTS = 500; NPOINTS <= 5000; NPOINTS += 500)
    {
        int i, j, iter, numoutside, value = 0;
        double area, error, ztemp;
        double start, finish;
        struct complex z, c;

        start = MPI_Wtime();

       #pragma omp parallel for private(c, z, ztemp, j, iter) schedule(guided)
        for (i = rank; i < NPOINTS; i += size)
        {
            for (j = 0; j < NPOINTS; j++)
            {
                c.real = -2.0 + 2.5 * (double)(i) / (double)(NPOINTS) + 1.0e-7;
                c.imag = 1.125 * (double)(j) / (double)(NPOINTS) + 1.0e-7;
                z = c;
                for (iter = 0; iter < MAXITER; iter++)
                {
                    ztemp = (z.real * z.real) - (z.imag * z.imag) + c.real;
                    z.imag = z.real * z.imag * 2 + c.imag;
                    z.real = ztemp;
                    if ((z.real * z.real + z.imag * z.imag) > 4.0e0)
                    {
                        #pragma omp critical
                        value++;
                        break;
                    }
                }
            }
        }

        MPI_Reduce(&value, &numoutside, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        finish = MPI_Wtime();

        if (rank == 0)
        {
            area = 2.0 * 2.5 * 1.125 * (double)(NPOINTS * NPOINTS - numoutside) / (double)(NPOINTS
* NPOINTS);
            error = area / (double)NPOINTS;

            printf("NPOINTS: %d | Area of Mandlebrot set = %12.8f +/- %12.8f\n", NPOINTS, area,
error);
            printf("Time = %12.8f seconds\n", finish - start);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Figura 3 – Código fonte da versão paralela em MPI e OpenMP (programa em C)

3.3 Balanceamento de carga

Como o laço mais interno contém o código *break*, existe a possibilidade de algumas *threads* acabarem antes das outras, desta maneira é possível realizar o balanceamento de carga, aplicando em conjunto com a instrução *#pragma omp parallel for private(c, z, ztemp, j, iter)*, uma nova instrução, *schedule(guided).*

```
#pragma omp parallel for private(c, z, ztemp, j, iter)schedule(guided)
```

Figura 4 – Trecho de código fonte da versão paralela com balanceamento (programa em C)

4 RESULTADOS

Após a etapa da implementação e obtenção dos resultados utilizando a máquina Grad da PUCRS, a qual já foi mencionada na seção 1 (Introdução), com os dados em mãos foi possível gerar os gráficos para melhor ilustração da diferença de se paralelizar uma aplicação. Os gráficos a seguir mostram o tempo de execução das versões do algoritmo em relação aos NPOINTS a serem realizados. Onde cada uma dessas versões o que muda é a quantidade de nodos deslocados para execução do algoritmo e com isso os gráficos se comportaram da seguinte forma:

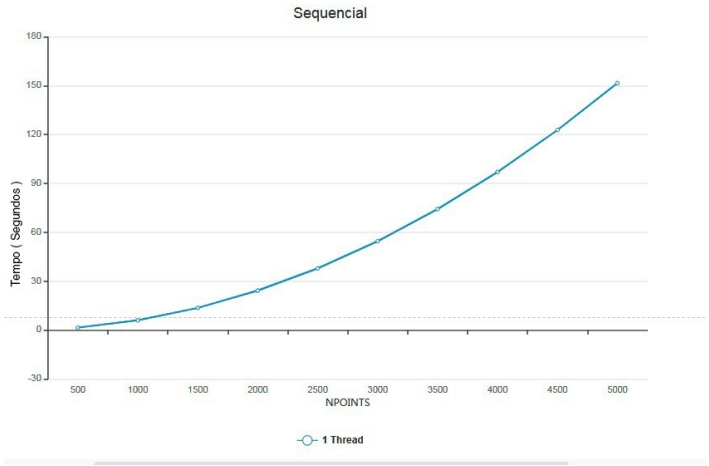

Figura 5 – Resultados com a versão sequencial

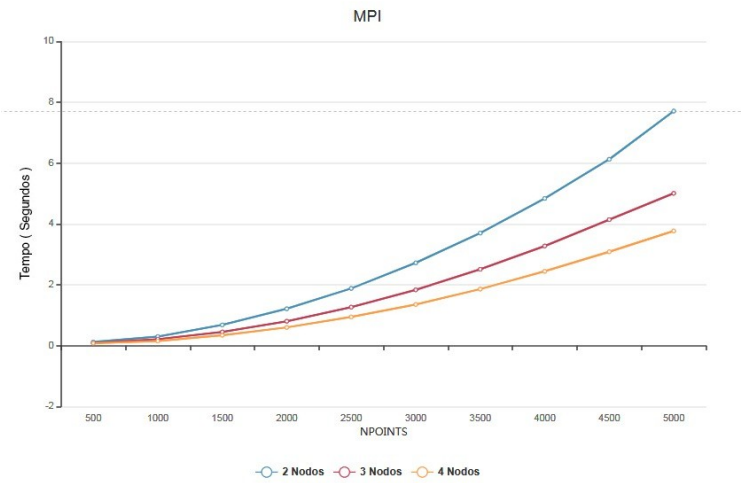

Figura 6 – Resultados com MPI

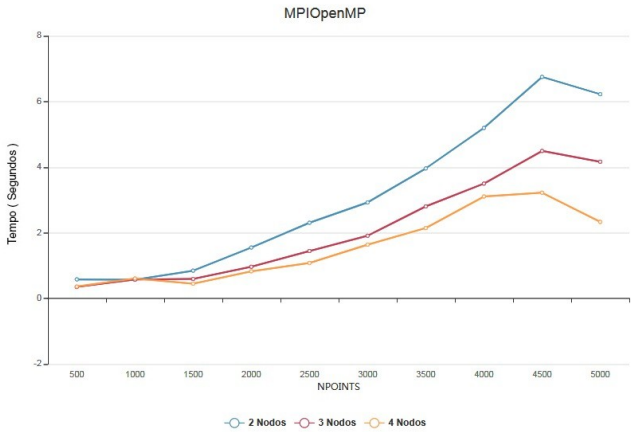

Figura 7– Resultados MPI e OpenMP

Para fins de comparação a versão sequencial do algoritmo demorou em torno de 150 segundos para obter os resultados, com a versão utilizando MPI o resultado foi em torno de 3 segundos e meio e com MPI e OpenMP o resultado foi em torno de 2 segundos e meio. Com NPOINTS igual a 5000 e as duas últimas execuções citadas usando os 4 nodos com todas as *threads.*

Dito isso, o gráfico a seguir, irá apresentar a medida de eficiência do *Speed-Up* em comparação ao *Speed-Up* ideal com relação ao número de *threads*:

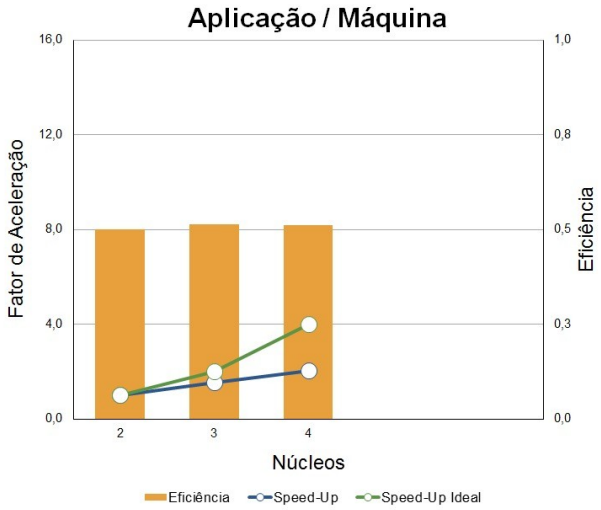

Figura 8 – SpeedUp com MPI

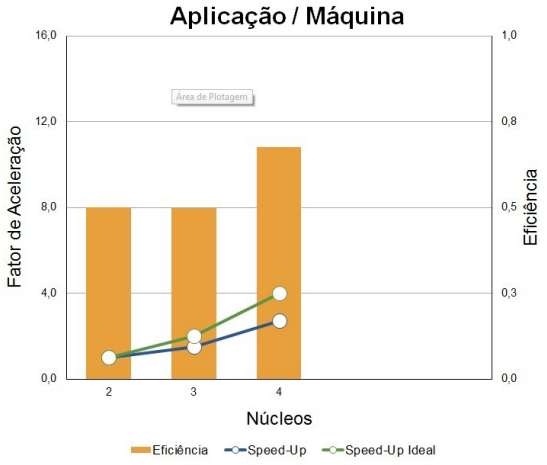

Figura 9 – SpeedUp com MPI e OpenMP

Observando as imagens acima, pode-se afirmar que com 2 e 3 nodos a eficiência em ambos algoritmos é quase a ideal. Já *com a utilização de 4 nodos* o ganho de desempenho não é tão impactante com a quantidade maior de processadores. Apesar disso a execução do código com MPI e OpenMP ainda apresenta um melhor *SpeedUp* com 4 nodos quando comparado com a versão com apenas MPI.

5 Conclusão

Este artigo apresentou sugestões de paralelismo para o código sequencial da Área de um Conjunto de Mandelbrot (BULL, 2019). Além disto também apresentamos os resultados obtidos juntos com MPI e MPI em conjunto com *OpenMP*, compreendendo as métricas de avaliação do desempenho de um código paralelo. Estimamos que utilizamos em torno de 4 horas dos nodos do LAD para todos os testes.

Podemos concluir que a paralelização impactou de forma positiva e drástica no tempo de execução do algoritmo mesmo com adição de poucas linhas.

Referências

BULL, Mark. Shared Memory Programming with OpenMP - Exercise Notes. [S.l.]: ARCHER, 11 nov. 2019.

Disponível em:

< http://www.archer.ac.uk/training/courses/2019/11/openmp_online/OpenMP-Online-Exercises.pdf >. Acesso em 19 maio 2020.

CONJUNTO DE MANDELBROT. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation,

2020. Disponível em: < https://pt.wikipedia.org/wiki/Conjunto_de_Mandelbrot >. Acesso em: 19 maio 2020

Message Passing Interface. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation,

2020. Disponível em: <https://pt.wikipedia.org/wiki/Message_Passing_Interface>. Acesso em 23 maio 2020