Trabalho Prático 2: Mandelbrot com MPI

Jecé Xavier Pereira Neto - 52552,

Matheus Rigoni Galvão - 54185

Supervisors:

Profº. Dr. José Rufino

Bragança

2022-2023

Conteúdo

1 Introdução **1**

1.1 Objetivos 1

1.2 Estrutura do Documento 1

2 Metodologia **2**

2.1 Profiling com Valgrind 2

2.2 Cálculo de Speedup e Eficiência teóricos 5

2.3 Paralelização com MPI 5

3 Testes e Resultados **10**

4 Conclusão e Discussão **15**

Lista de Tabelas

2.1 Speedups calculados pela Lei de Amdahl 5
2.2 Eficiências calculadas usando o Speedup téorico 5

3.1 Tempos reais da versão MPI 11
3.2 Speedup real da versão MPI 11
3.3 Eficiência da versão paralela e proximidade com eficiência teórica (%) 12

Lista de Figuras

2.1 Tabela de principais hotspots 3
2.2 Call graph main 4
2.3 Função main 6
2.4 Broadcast de variáveis 7
2.5 Cálculo de linhas para cada processo 8
2.6 Envio de cálculos para o processo 0 9
2.7 Finalização do processo 0 9

3.1 Função screen_dump 10
3.2 Gráfico de tempo real 11
3.3 Gráfico de speedup real 12
3.4 Gráfico de eficiência real 13
3.5 Gráfico de ratio entre eficiência real e teórica 13
3.6 Imagem final da versão serial 14
3.7 Imagem final da versão MPI com 8 processos 14

Capítulo 1

Introdução

O presente trabalho relata o desenvolvimento de uma versão paralela do algoritmo gerador do set de Mandelbrot, através de programação *Message Passing Interface* (MPI). O relatório contém a metodologia utilizada para identificar *hotspots* da versão serial, cálculos de *speedup* e eficiência teóricos, descrição da estratégia seguida para integrar MPI ao código serial e como a comunicação é realizada, testes e resultados obtidos, e por fim as conclusões feitas durante o trabalho são apresentadas e discutidas.

1.1 Objetivos

Otimizar o algortimo de geração do set de Mandelbrot atráves do uso de programação MPI, realizar testes e avaliar os resultados obtidos.

1.2 Estrutura do Documento

Este documento está estruturado em capítulos. O capítulo 1 se refere a introdução, os objetivos deste trabalho, e a estrutura do documento. O capítulo 2 descreve a metodologia utilizada para solucionar o problema. O capítulo 3 contém os resultados dos testes realizados. Por final, o capítulo 4 apresenta as conclusões obtidas e discute sobre possíveis melhoras em trabalhos futuros.

Capítulo 2

Metodologia

Este capítulo é dedicado a descrição do processo de desenvolvimento do projeto, desde o uso de ferramentas de profiling, juntamente com o comando utilizado para o uso, identificação de principais hotspots do algoritmo, cálculos téoricos de Speedup e eficiência apresentados em tabelas, e uma descrição detalhada acompanha de imagens das alterações realizadas no código para a integração do MPI.

2.1 Profiling com Valgrind

A realização do profiling do algoritmo foi realizado usando a ferramenta callgrind, através do seguinte comando para a versão serial.

valgrind –tool=callgrind ./mandelbrot-gui-serial.exe

Os resultados visualizados através do programa KCachegrind para determinar os hotspots são apresentados na Figura 2.1. O maior hotspot observado é referente a função calc_mandel, a qual constitui 98.78% do processamento total, tal função é responsável por calcular o valor de cada píxel da tela através de laços de repetição para linhas e colunas, os quais podem ser paralelizados por meio de uma divisão de setores.

O call graph da função main é apresentado na Figura 2.2.

Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000020…	ld-linux-x86-64.so.2
100.00	0.00	1	(below main)	mandelbrot-gui-serial.exe
100.00	0.00	1	__libc_start_main@…	libc.so.6: libc-start.c
100.00	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
100.00	0.00	1	main	mandelbrot-gui-serial.exe: mandelbrot-gui-serial.c
99.57	0.00	20	set_texture	mandelbrot-gui-serial.exe: mandelbrot-gui-serial.c
99.55	0.00	1	0x0000000000109…	(unknown)
99.55	0.00	1	glutMainLoop	libglut.so.3.9.0
99.54	0.00	3	glutMainLoopEvent	libglut.so.3.9.0
98.78	91.13	20	calc_mandel	mandelbrot-gui-serial.exe: mandelbrot-gui-serial.c
98.49	0.00	1	keypress	mandelbrot-gui-serial.exe: mandelbrot-gui-serial.c
98.49	0.00	18	mouseclick	mandelbrot-gui-serial.exe: mandelbrot-gui-serial.c
5.88	3.02	14 942 208	hsv_to_rgb	mandelbrot-gui-serial.exe: mandelbrot-gui-serial.c
2.86	0.12	29 884 416	0x0000000000109…	(unknown)
2.74	0.75	29 884 416	fmod	libm.so.6: w_fmod_compat.c
1.99	1.99	29 884 416	__fmod_finite@GLI…	libm.so.6: e_fmod.c
1.77	0.06	14 942 208	0x0000000000109…	(unknown)
1.71	1.71	14 942 208	hypot@@GLIBC_2.35	libm.so.6: e_hypot.c
1.05	0.00	1	resize	mandelbrot-gui-serial.exe: mandelbrot-gui-serial.c
0.75	0.00	20	0x0000000000109…	(unknown)
0.75	0.00	20	0x000000000000e…	libglapi.so.0.0.0
0.75	0.00	20	0x000000000013e…	iris_dri.so
0.75	0.00	20	0x000000000013b…	iris_dri.so
0.75	0.00	20	0x0000000000176…	iris_dri.so
0.75	0.00	20	0x0000000000174c…	iris_dri.so
0.75	0.00	20	0x0000000000152…	iris_dri.so
0.75	0.00	20	0x0000000000151…	iris_dri.so
0.75	0.00	20	0x00000000002cbe…	iris_dri.so
0.75	0.75	19 457	0x00000000002c38…	iris_dri.so
0.45	0.00	1	init_gfx	mandelbrot-gui-serial.exe: mandelbrot-gui-serial.c
0.42	0.00	1	0x0000000000109…	(unknown)
0.42	0.00	1	glutCreateWindow	libglut.so.3.9.0
0.42	0.00	1	fgCreateWindow	libglut.so.3.9.0
0.42	0.00	1	fgOpenWindow	libglut.so.3.9.0
0.37	0.00	1	fgChooseFBConfig	libglut.so.3.9.0

Figura 2.1: Tabela de principais hotspots

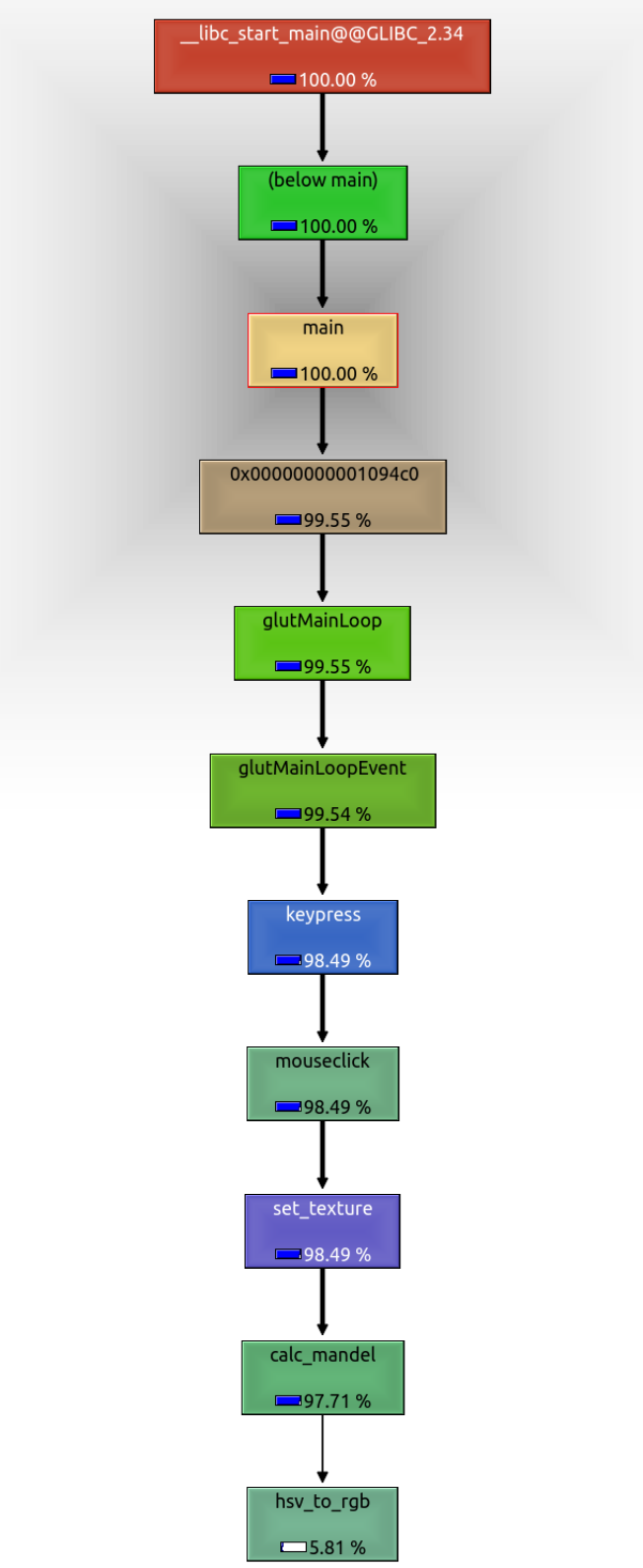

Figura 2.2: Call graph main

2.2 Cálculo de Speedup e Eficiência teóricos

Considerando a fração paralelizável do algoritmo como 98.78%, é possível calcular o Speedup téorico utilizando a Lei de Amdhal para um número de processos N=1...8 como apresentado na Tabela 2.1. É possível constatar um aumento de aceleração a cada novo processo utilizado, porém o aumento é cada vez menor devido ao fato de a Lei de Amdahl levar em conta a porção não paralelizavel do código, a qual gera um limite de speedup.

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
St	1	1.98	2.93	3.86	4.77	5.66	6.52	7.37

Tabela 2.1: Speedups calculados pela Lei de Amdahl

Após o cálculo do speedup para cada quantidade de threads, a eficiência é calculada pela divisão do speedup por seu respectivo número de processos. Os resultados de tais cálculos são apresentados na Tabela 2.2.

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Et(%)	100	99	98	96	95	94	93	92

Tabela 2.2: Eficiências calculadas usando o Speedup téorico

2.3 Paralelização com MPI

Com o objetivo de desenvolver uma versão do código utilizando MPI para executar o algoritmo de forma paralela, diversas alterações foram feitas ao código serial, as quais são descritas acompanhandas por trechos de código referentes as respectivas alterações.

A primeira etapa tem como objetivo determinar quais funções cada processo deve executar, a lógica desenvolvida utiliza o *rank* do processo para fazer tal organização, a qual pode ser visualizada na Figura 2.3. Todas as funções originais da função main são executadas no processo 0, incluindo o a função calc_mandel, a qual é executada em outra parte do código, os demais processos executam exclusivamente a função calc_mandel dentro de um laço por 20 repetições, tal número foi observado como sendo o número de vezes que a função é executada na versão serial.

```c
int main(int c, char **v)
{

    MPI_Init(&c, &v);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank == 0) // Task 0 executa tudo, incluindo o calc mandel
    {
        init_gfx(&c, v, rank);
        print_menu();
        glutMainLoop();
    } else { // Outras tasks executam só o calc mandel
        for(int c=0; c<20; c++)
            calc_mandel();

    }

    MPI_Finalize();

    return 0;

}
```

Figura 2.3: Função main

A função calc_mandel, como citada anteriormente, é responsável por calcular o valor de cada píxel da tela, mas antes de qualquer cálculo ser feito, é necessário atualizarmos todas as variáveis globais de cada processo. Por meio de vetores, todas as variáveis de tipo integer e double são encapsuladas (função into_arrays) pelo processo 0, enviadas por broadcast para todos os processos, e desencapsuladas (função out_arrays) para cada variável global, assim todas as variáveis globais são enviadas através de dois broadcasts. Em seguida, outro broadcast é feito, o qual envia todos os bytes do ponteiro GLOBAL_tex, porém quando recebidos pelos outros processos, os endereços referentes ao primeiro pixel de cada linha não fazem sentido na memória de cada processo, sendo necessário recalcular tais endereços. Toda a lógica citada acima pode ser vista na Figura 2.4.

```
void calc_mandel()
{
    int i;

    if(rank == 0)
    {
        into_arrays(); // Passa todas as variaveis para dois arrays
    }

    MPI_Bcast(&bc_array_int, 12, MPI_INT, 0, MPI_COMM_WORLD); // Manda todos os ints
    MPI_Bcast(&bc_array_double, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD); // Manda todos os doubles

    if(rank != 0)
    {
        out_arrays(); // Passa todos os elementos dos arrays para as variaveis
        GLOBAL_tex = realloc(GLOBAL_tex, GLOBAL_tex_size); // Alocar tamanho pro global_tex
    }
    MPI_Bcast(GLOBAL_tex, GLOBAL_tex_size, MPI_BYTE, 0, MPI_COMM_WORLD); // Manda todos os bytes do GLOBAL_tex


    if(rank != 0) // Recalcular enderecos das linhas
    {
        for (GLOBAL_tex[0] = (rgb_t *)(GLOBAL_tex + GLOBAL_tex_h), i = 1; i < GLOBAL_tex_h; i++)
            GLOBAL_tex[i] = GLOBAL_tex[i - 1] + GLOBAL_tex_w;
    }
```

Figura 2.4: Broadcast de variáveis

Ainda na função calc_mandel, a divisão da carga de trabalho para cada processo deve ser feita, o método escolhido foi a divisão igual e sequencial de linhas para cada processo, através da definição de um limite inferior e outro superior, assim o processo 0 processa certa quantidade de linhas, o processo 1 a mesma quantidade de linhas a seguir, e tal lógica é seguida até o último processo. Contudo, o número de processos deve ser par para

realizar uma divisão igual e integral de todas as linhas, caso contrário todos os processos terão a mesma carga de trabalho, porém algumas linhas não serão processadas, cerca de 3 a 5. O código referente aos cálculos dos píxeis não foi alterado, exceto pelo laço de repetição que controla as linhas, como pode ser visto na Figura 2.5.

```
// Quantas cada task vai fazer (só pode ser par)
int linhas_cada = GLOBAL_height/numtasks;
int lim_inf = rank*linhas_cada;
int lim_sup = lim_inf + linhas_cada;

// Calcula o px, min, max
for (i = lim_inf; i < lim_sup; i++) {⋯

// Passa min, max e cada px para a tela
for (i = lim_inf; i < lim_sup; i++)
    for (j = 0, px = GLOBAL_tex[i]; j  < GLOBAL_width; j++, px++)
        hsv_to_rgb(*(unsigned short*)px, min, max, px);
```

Figura 2.5: Cálculo de linhas para cada processo

Após os cálculos terem sido realizados, cada processo terá uma versão do GLOBAL_tex com uma porção de píxeis calculados corretamente, sendo necessário o envio de cada porção correta para o processo 0, o qual é encarregado de atualizar a janela. O método escolhido para o envio das porções foi o MPI_Gather, no qual todos os píxeis corretos são passados para um buffer de envio, os quais serão recebidos sequencialmente no processo 0 pelo GLOBAL_tex. Toda a sequência de operações descritas anteriormente está contida na Figura 2.6.

Por alguma razão não compreendida, após a destruição da janela no processo 0, tal processo não retorna à função main, consequentemente não executando o MPI_Finalize, para resolver tal problema, a inserção de um MPI_Finalize após a chamada do tecla 'q' foi necessária, como pode ser visto na Figura 2.7.

```
// Buffer para envio
rgb_t *sendbuf = 0;

// Tamanho do buffer a ser enviado
int SEND_BUFFER_SIZE = linhas_cada * GLOBAL_tex_w * sizeof(rgb_t);
// Espaco alocado para o buffer de envio
sendbuf = realloc(sendbuf, SEND_BUFFER_SIZE);

int aux = 0;
// Preenche o buffer de envio com todos os pixeis que a task calculou
for (i = lim_inf; i < lim_sup; i++)
{
    for (j = 0, px = GLOBAL_tex[i]; j < GLOBAL_width; j++, px++)
    {
        sendbuf[aux] = *px;
        aux++;
    }
}

// Envia cada pedaco do GLOBAL_tex calculado para a task 0
MPI_Gather(sendbuf, SEND_BUFFER_SIZE, MPI_BYTE, *GLOBAL_tex, SEND_BUFFER_SIZE, MPI_BYTE, 0, MPI_COMM_WORLD);
```

Figura 2.6: Envio de cálculos para o processo 0

```
case 'Z':// simulate many mouse clicks in order to dive fully in zoomin
        // COMEÇAR A CONTAR O TEMPO
        gettimeofday(&start,NULL);

         GLOBAL_refresh=1; // use 0 to avoid refreshing all but the last one
         for (zoomin_x=0, zoomin_y=1; zoomin_x < GLOBAL_zoomin_num_pairs; zoomin_x+=2, zoomin_y +=2) {
            if (zoomin_x == GLOBAL_zoomin_num_pairs-2) GLOBAL_refresh=1;

            mouseclick(GLUT_LEFT_BUTTON, GLUT_UP, GLOBAL_zoomin[zoomin_x], GLOBAL_zoomin[zoomin_y]);
         }
         // simulate case 's'
         keypress('s', -1, -1);
         // simulate case 'q'
         keypress('q', -1, -1);

         MPI_Finalize();
         return;
}
```

Figura 2.7: Finalização do processo 0

Capítulo 3

Testes e Resultados

Em prol de realizar os testes necessários, o tempo de execução do programa teve de ser calculado em microsegundos, sendo iniciado logo após o usuário selecionar a tecla 'Z', como pode ser visto na Figura 2.7, e finalizado no início da função screen_dump, presente na Figura 3.1.

```
void screen_dump()
{
    // TERMINAR DE CONTAR O TEMPO E PRINTAR
    gettimeofday(&end,NULL);
    int elapsed = ((end.tv_sec - start.tv_sec) * 1000000) + (end.tv_usec - start.tv_usec);
    printf("\nTime elapsed: %d micro seconds\n",elapsed);

    static int dump=1;
    char fn[100];
    int i;
    sprintf(fn, "screen%03d-mpi.ppm", dump++);
    FILE *fp = fopen(fn, "w");
    fprintf(fp, "P6\n%d %d\n255\n", GLOBAL_width, GLOBAL_height);
    for (i = GLOBAL_height - 1; i >= 0; i--)
        fwrite(GLOBAL_tex[i], 1, GLOBAL_width * 3, fp);
    fclose(fp);
    printf("%s written\n", fn);
}
```

Figura 3.1: Função screen_dump

Com o código para calcular o tempo de execução integrado em ambas as versões serial e paralelizada, o testes de tempo para cada quantidade de processos foram realizados

utilizando a configuração de benchmarking no makefile e GLOBAL_max_iter = 4096. Todos os tempos de execução foram medidos 3 vezes e o menor tempo foi selecionado, a máquina utilizada para testes contém uma CPU Intel i5 8th Gen e 8GB de memória RAM. A versão serial do código resultou em 18118728 microsegundos, cerca de 18,11 segundos, já os tempos reais de execução da versão MPI estão contidos na Tabela 3.1, acompanhados pelo gráfico presente na Figura 3.2. O speedup real para cada número de processos obtido pela divisão do tempo serial pelo tempo paralelo pode ser visualizado na Tabela 3.2 e no gráfico contido na Figura 3.3.

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Tr(S)	18.14	15.62	12.71	10.66	9.70	8.67	7.16	6.54

Tabela 3.1: Tempos reais da versão MPI

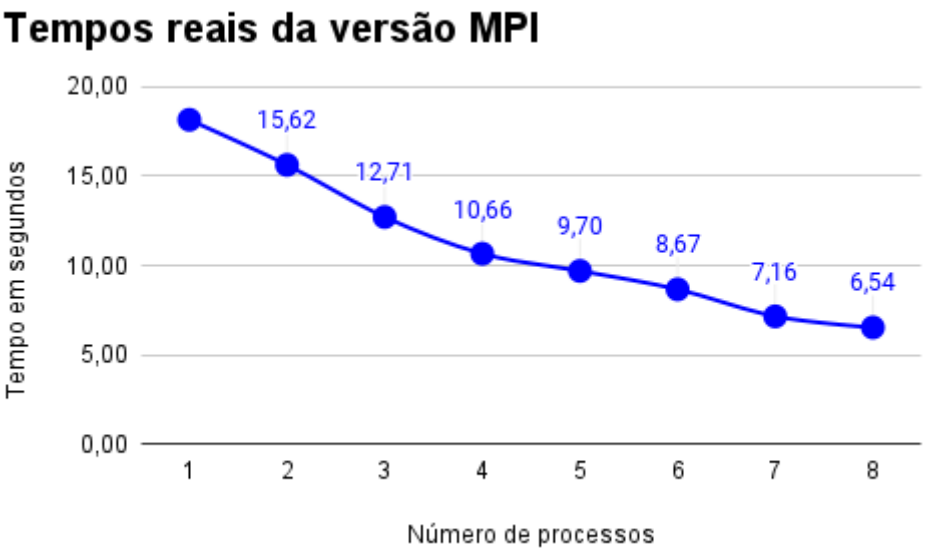

Figura 3.2: Gráfico de tempo real

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Sr	-	1.15	1.42	1.69	1.86	2.08	2.52	2.76

Tabela 3.2: Speedup real da versão MPI

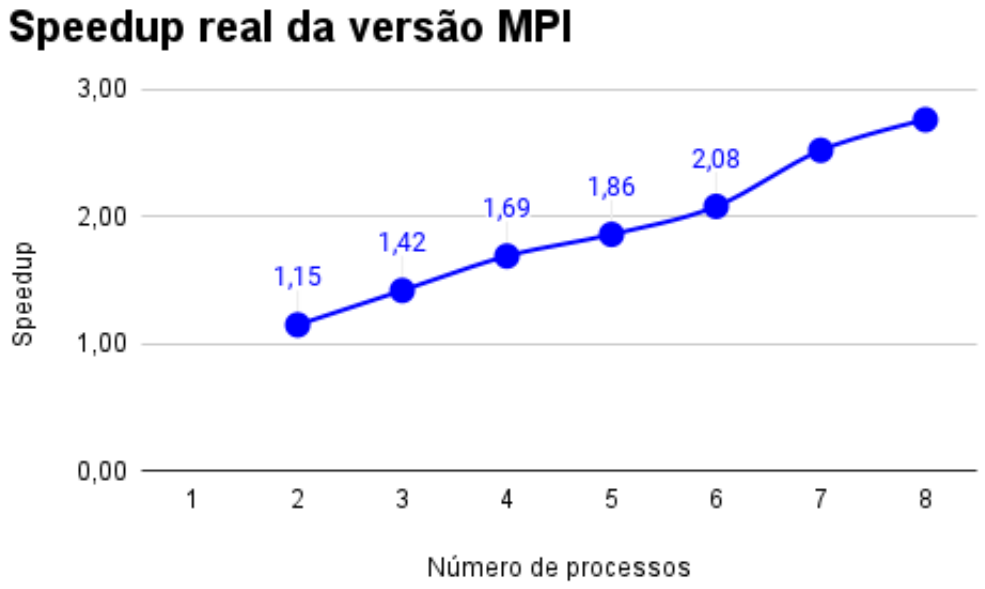

Figura 3.3: Gráfico de speedup real

A Tabela 3.3 contém as eficiências reais em porcentagem obtidas da versão MPI, também contidas na Figura 3.4, acompanhadas de uma métrica para avaliar o quão perto a versão paralelizada chegou das previsões realizadas com a Lei de Amdahl, tal métrica é constituída pela divisão da eficiência real pela eficiência teórica e possui um gráfico próprio presente na Figura 3.5. A fins de comparação, as imagens resultantes da execução da versão serial e da versão MPI com 8 processos podem ser visualizadas nas Figuras 3.6 e 3.7 respectivamente.

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Er(%)	-	58	47	42	37	35	36	35
Er/Et(%)	-	59	49	44	39	37	39	38

Tabela 3.3: Eficiência da versão paralela e proximidade com eficiência teórica (%)

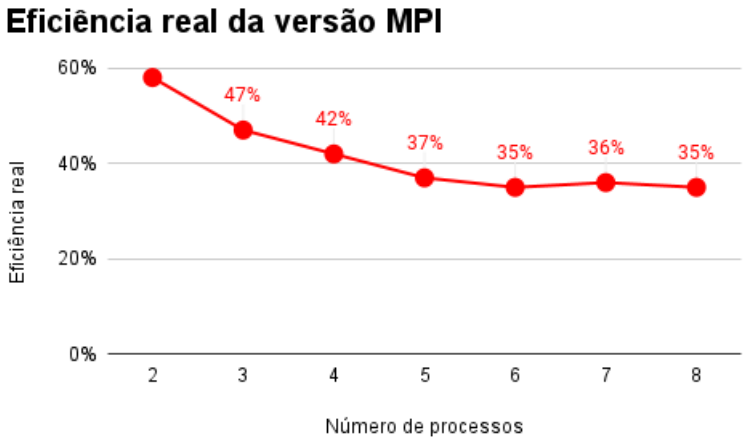

Figura 3.4: Gráfico de eficiência real

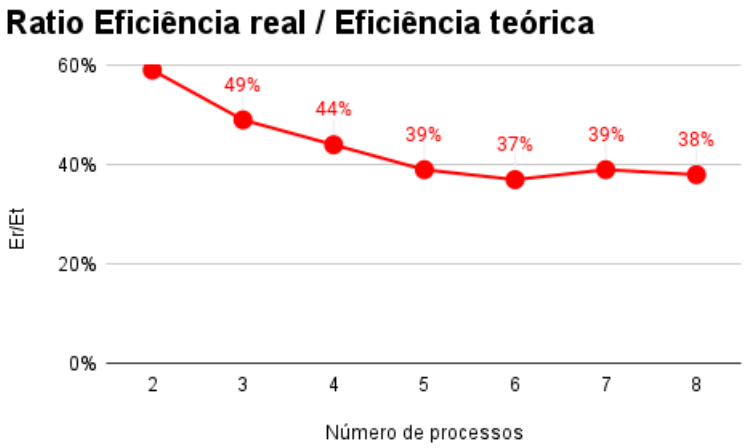

Figura 3.5: Gráfico de ratio entre eficiência real e teórica

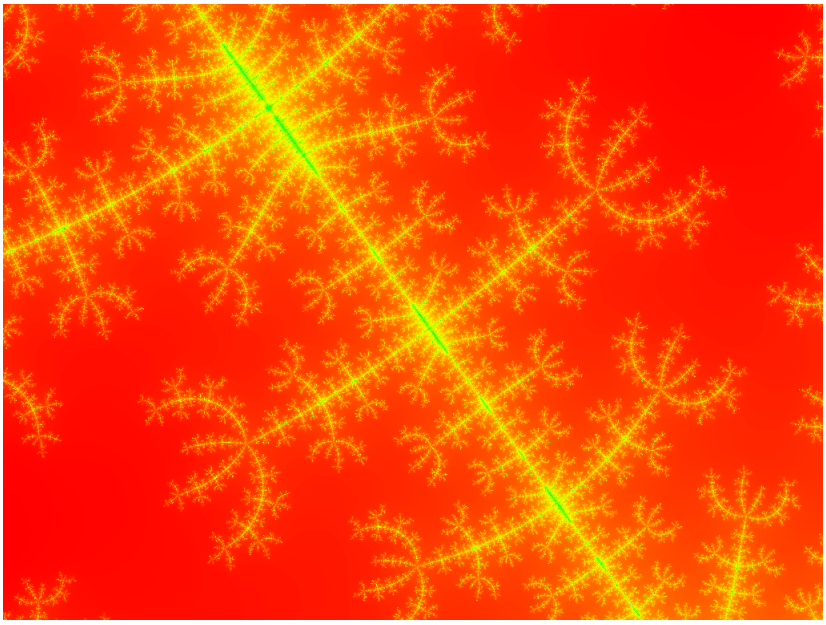

Figura 3.6: Imagem final da versão serial

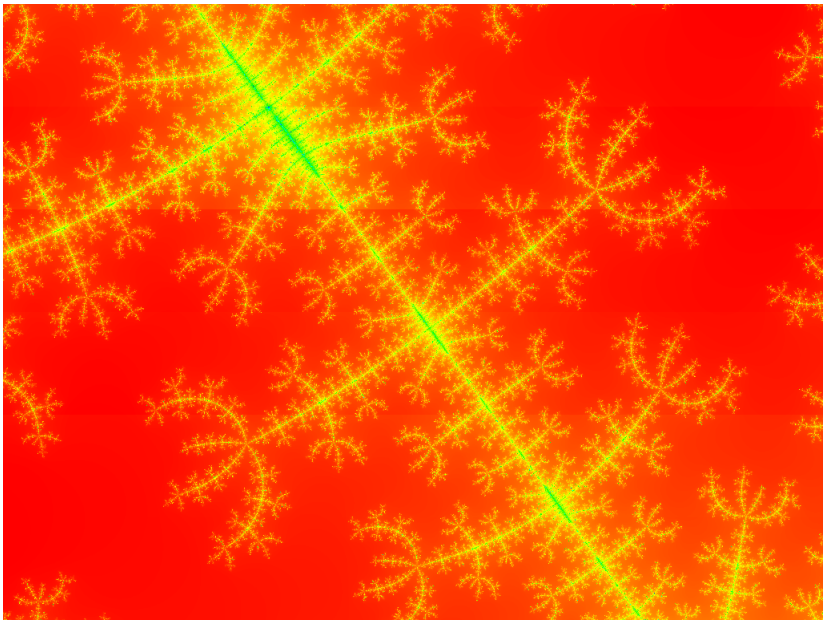

Figura 3.7: Imagem final da versão MPI com 8 processos

Capítulo 4

Conclusão e Discussão

Os resultados obtidos mostram claramente uma melhora no tempo de execução diretamente proporcional ao número de processos utilizados, porém os valores reais de speedup e eficiência são significativamente menores que os valores téoricos previamente calculados. Tal diferença existe pelo fato de os cálculos baseados na Lei de Amdahl não considerarem os diversos tipos de overhead criados na versão paralelizada com MPI, tais como barreiras e comunicações coletivas, as quais exigem que os processos esperem uns aos outros, como também operações necessárias para organizar o funcionamento de variáveis e laços de repetições. O código pode ser futuramente aperfeiçoado com algumas alterações para um melhor funcionamente e robustez, entre elas estão a minimização de cópias de dados, especialmente no processo 0, o uso do parâmetro MPI_IN_PLACE pode ser uma solução viável para tal problema, o uso de tipos derivados de dados é uma alternativa a ser analisada e testada para simplificar a comunicação de diversas variáveis e tornar o sistema mais robusto e escalável. Apesar das possíveis melhoras ainda a serem feitas, os objetivos definidos para o projeto foram alcançados, com a implementação bem sucedida de programação MPI ao algoritmo de geração do set de Mandelbrot, a qual tornou o processamento mais rápido e eficiente.