

Enunciado do EP3

Informações Gerais

Calculando um Conjunto de Julia

Tarefa 1 - Versão Sequencial

Primeira Parte - Computação do Conjunto

Segunda Parte - Salvando num Arquivo

Tarefa 2 - Versão Paralela

Primeira Parte - Distribuição dos Dados

Segunda Parte - Computação do Conjunto Usando MPI

Terceira Parte - Salvando num Arquivo

Tarefa 3 - Experimentos

Experimentos com MPICH

Experimentos com SimGrid

Informações Gerais

Título: EP3 - Mais Fractais

Disciplina: MAC0219/5742 - Programação Concorrente e Paralela - 2024

Autores: Lucas de Sousa Rosa e Alfredo Goldman

Disponibilizado: @24 de novembro de 2024

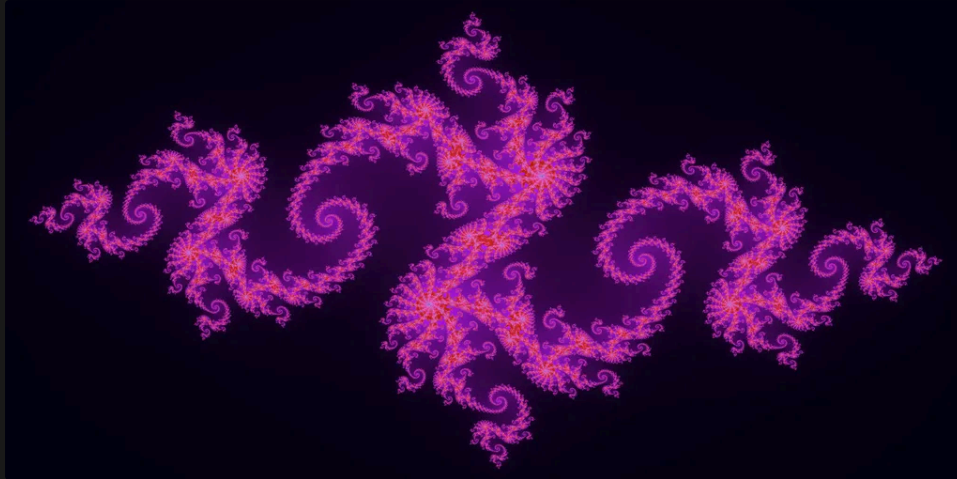
Primeira Data de Entrega: @11 de dezembro de 2024



Este EP é baseado no tutorial [🔗 Topic #1 Module · SMPI CourseWare](#). Para instalar o SimGrid siga as instruções em [🔗 Topic #0 Module · SMPI CourseWare](#). Instale também o [MPICH](#) na máquina de vocês.

Todos os artefatos referentes às entregas de cada tarefa deverão ser reunidos em um único arquivo compactado (preferencialmente nos formatos .zip ou .tar.gz) e enviados por meio do e-Disciplinas.

Calculando um Conjunto de Julia



Um conjunto de Julia é um tipo de fractal formado ao iterar uma função matemática envolvendo números complexos, como $f(z) = z^2 + c$. Ele mostra os pontos no plano complexo que permanecem dentro de limites definidos após várias iterações. A aparência do conjunto muda conforme o valor de c , criando padrões simétricos e detalhados que se repetem em diferentes escalas.

O conjunto particular de Julia que vamos calcular aqui é definido como segue. Dado z um ponto no plano complexo 2D, calculamos a série definida como:

- $z_0 = z$
- $z_{n+1} = z_n^2 + c$

onde $c = -0,79 + 0,15i$. Valores diferentes de c levam a imagens diferentes, e conhecemos muitos valores que produzem imagens bonitas.

A cor de um pixel correspondente ao ponto z é determinada com base no número de iterações antes que o módulo de z_n fique acima de 2 (ou até que um número máximo de iterações seja atingido). Não se preocupe que o código que calcula e colore é fornecido.

Tarefa 1 - Versão Sequencial

Primeira Parte - Computação do Conjunto

A primeira parte desta tarefa é criar um programa sequencial simples que calcula um conjunto de Julia e salva o resultado em um arquivo bitmap.

Implemente um programa em C (ou C++) chamado `sequential_julia.c`, que deverá:

- Receber um único argumento na linha de comando, `n`, um inteiro estritamente positivo.

- Alocar um array unidimensional de `unsigned char` com $6n^2$ elementos. Este array será usado para armazenar os pixels de uma imagem com altura n e largura $2n$, onde cada pixel é representado por 3 bytes (valores RGB).
- Preencher este array com os valores correspondentes aos pixels do conjunto de Julia.

Para facilitar, utilize a função `compute_julia_pixel()` fornecida. Ela recebe as coordenadas (x, y) de um pixel, as dimensões (largura e altura) da imagem, um valor `float tint` (use `1.0` por enquanto) e um ponteiro para 3 bytes contíguos (`unsigned char *`). Esses bytes serão configurados com os valores RGB apropriados para exibir o conjunto de Julia.

► **Função:** `compute_julia_pixel()` (use a seta para revelar o código)

O programa deve armazenar os pixels da imagem 2D na matriz 1D usando um esquema **row-major**.

► Mais detalhes sobre o esquema row-major.

Segunda Parte - Salvando num Arquivo

Modifique seu programa para que ele salve a imagem em um arquivo chamado `julia.bmp`, que armazena a imagem no formato BMP. Um arquivo BMP consiste em duas partes:

- Cabeçalho
- Os pixels de 3 bytes da imagem no esquema row-major

Para facilitar o procedimento acima, fornecemos outra função auxiliar C, `write_bmp_header()`, que grava em um arquivo o cabeçalho de 40 bytes necessário para um arquivo BMP.

▼ Função: `write_bmp_header()`

```

/* write_bmp_header(): * * Entrada: * f: Um arquivo aberto para
escrita ('w') * (width, height): dimensões da imagem * * Retorno: * 0
em caso de sucesso, -1 em caso de falha * */ int write_bmp_header(FILE
*f, int width, int height) { unsigned int row_size_in_bytes = width *
3 + ((width * 3) % 4 == 0 ? 0 : (4 - (width * 3) % 4)); // Define
todos os campos no cabeçalho do BMP char id[3] = "BM"; unsigned int
filesize = 54 + (int)(row_size_in_bytes * height * sizeof(char));
short reserved[2] = {0,0}; unsigned int offset = 54; unsigned int size
= 40; unsigned short planes = 1; unsigned short bits = 24; unsigned
int compression = 0; unsigned int image_size = width * height * 3 *
sizeof(char); int x_res = 0; int y_res = 0; unsigned int ncolors = 0;
unsigned int importantcolors = 0; // Escreve os bytes no arquivo,
mantendo o controle do // número de "objetos" escritos size_t ret = 0;
ret += fwrite(id, sizeof(char), 2, f); ret += fwrite(&filesize,
sizeof(int), 1, f); ret += fwrite(reserved, sizeof(short), 2, f); ret
+= fwrite(&offset, sizeof(int), 1, f); ret += fwrite(&size,
sizeof(int), 1, f); ret += fwrite(&width, sizeof(int), 1, f); ret +=
fwrite(&height, sizeof(int), 1, f); ret += fwrite(&planes,
sizeof(short), 1, f); ret += fwrite(&bits, sizeof(short), 1, f); ret
+= fwrite(&compression, sizeof(int), 1, f); ret += fwrite(&image_size,
sizeof(int), 1, f); ret += fwrite(&x_res, sizeof(int), 1, f); ret +=
fwrite(&y_res, sizeof(int), 1, f); ret += fwrite(&ncolors,
sizeof(int), 1, f); ret += fwrite(&importantcolors, sizeof(int), 1,
f); // Sucesso significa que escrevemos 17 "objetos" com êxito return
(ret != 17); }

```

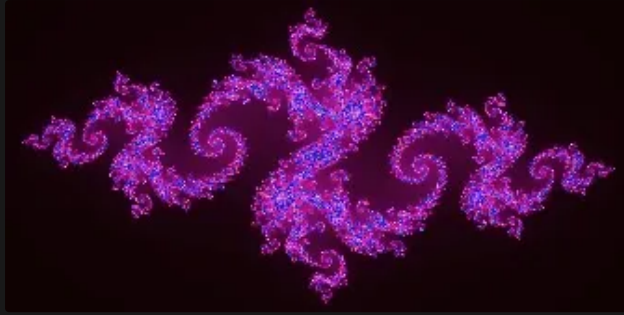
O fragmento de código abaixo mostra como gravar os pixels no arquivo bpm (aberto como arquivo `output_file`) depois que o cabeçalho foi gravado:

```

// Escrevendo os pixels após o cabeçalho for (y=0; y < height; y++) { for
(x=0; x < width; x++) { fwrite(&(pixels[y * 3 * width + x * 3]),
sizeof(char), 3, output_file); } // padding no caso de um número par de
pixels por linha unsigned char padding[3] = {0,0,0}; fwrite(padding,
sizeof(char), ((width * 3) % 4), output_file); }

```

Com isso, deve ser simples modificar seu programa para que ele produza um arquivo BMP do conjunto Julia. Verifique se a imagem produzida é como mostrada abaixo (abrindo em qualquer aplicativo de visualização de imagens ou em um navegador da Web):



Sua próxima tarefa será paralelizar o código sequencial com MPI.



Primeira Entrega (20%)

Código sequencial funcional `sequential_julia.c` + `Makefile` com o alvo para compilar.

Tarefa 2 - Versão Paralela

Primeira Parte - Distribuição dos Dados

Ao projetar um programa de memória distribuída, o primeiro passo é definir como os dados (entrada e/ou saída) serão divididos entre os processos participantes, o que chamamos de **distribuição de dados**. No caso do programa para o conjunto de Julia, a única informação relevante a ser distribuída é a imagem de saída.

Como a imagem utiliza um esquema de armazenamento row-major (onde as linhas são armazenadas sequencialmente), faz sentido dividir os dados em faixas horizontais. Essa imagem possui n linhas de pixels, sendo n o valor fornecido como argumento na linha de comando. Antes de calcular o conjunto de Julia, é necessário determinar como essas linhas serão distribuídas entre os processos. Cada processo, baseado em seu ranque, identificará as linhas específicas pelas quais será responsável.

Implemente um programa MPI C (ou C++) chamado `1D_parallel_julia.c` que:


- Receba um único argumento na linha de comando, `n`, um inteiro estritamente positivo.


- Cada processo imprime seu ranque (dentro do total de processos) e as linhas da imagem que ele deve calcular, indicadas como um intervalo. Neste estágio, o código não realizará nenhum cálculo real. Use os exemplos de execução abaixo como referência para o formato da saída.
 - A distribuição de dados deve ser feita em tantos slabs quantos forem os processos, atribuindo a cada processo a responsabilidade por um conjunto contínuo de linhas.

Para simplificar, **você pode assumir que o número de processos divide n** , de modo que todos os processos tenham o mesmo número de linhas para calcular.

Você deve garantir que a distribuição seja a mais equilibrada possível (no máximo uma diferença de uma linha entre dois processos). Isso pode ser feito usando matemática discreta para que tudo seja calculado por uma única fórmula, mas também pode ser feito programaticamente com um laço (ou seja, "contando manualmente").

Execute seu programa num cluster simulado de 64 nós. Para detalhes da execução, veja os resultados abaixo. Os arquivos usados na execução são:

 [simple_cluster.xml](#) 187.9KB

 [simple_cluster_hostfile.txt](#) 1.2KB

Verifique se seu código produz os mesmos resultados abaixo.

▼ Execução com 5 processos e $n = 1000$

```
% smpicc -O3 1D_parallel_julia.c -o 1D_parallel_julia % smpirun -np 5
-hostfile ./simple_cluster_hostfile.txt -platform ./simple_cluster.xml
./1D_parallel_julia 1000 [Process 0 out of 5]: I should compute pixel
rows 0 to 199, for a total of 200 rows [Process 1 out of 5]: I should
compute pixel rows 200 to 399, for a total of 200 rows [Process 2 out
of 5]: I should compute pixel rows 400 to 599, for a total of 200 rows
[Process 3 out of 5]: I should compute pixel rows 600 to 799, for a
total of 200 rows [Process 4 out of 5]: I should compute pixel rows
800 to 999, for a total of 200 rows
```

▼ Execução com 16 processos e $n = 100$

Observe que a distribuição é balanceada, o que significa que o número de linhas difere em no máximo 1 entre os processadores.

```
% smpicc -O3 1D_parallel_julia.c -o 1D_parallel_julia % smpirun -np 16
-hostfile ./simple_cluster_hostfile.txt -platform ./simple_cluster.xml
./1D_parallel_julia 100 [Process 0 out of 16]: I should compute pixel
rows 0 to 6, for a total of 7 rows [Process 1 out of 16]: I should
compute pixel rows 7 to 13, for a total of 7 rows [Process 2 out of
16]: I should compute pixel rows 14 to 20, for a total of 7 rows
[Process 3 out of 16]: I should compute pixel rows 21 to 27, for a
total of 7 rows [Process 4 out of 16]: I should compute pixel rows 28
to 33, for a total of 6 rows [Process 5 out of 16]: I should compute
pixel rows 34 to 39, for a total of 6 rows [Process 6 out of 16]: I
should compute pixel rows 40 to 45, for a total of 6 rows [Process 7
out of 16]: I should compute pixel rows 46 to 51, for a total of 6
rows [Process 8 out of 16]: I should compute pixel rows 52 to 57, for
a total of 6 rows [Process 9 out of 16]: I should compute pixel rows
58 to 63, for a total of 6 rows [Process 10 out of 16]: I should
compute pixel rows 64 to 69, for a total of 6 rows [Process 11 out of
16]: I should compute pixel rows 70 to 75, for a total of 6 rows
[Process 12 out of 16]: I should compute pixel rows 76 to 81, for a
total of 6 rows [Process 13 out of 16]: I should compute pixel rows 82
to 87, for a total of 6 rows [Process 14 out of 16]: I should compute
pixel rows 88 to 93, for a total of 6 rows [Process 15 out of 16]: I
should compute pixel rows 94 to 99, for a total of 6 rows
```

▼ Execução com 16 processos e $n = 12$

Observe que, como n é pequeno, alguns processos não têm nada a fazer. (Não nos importamos com o que esses processos imprimem como seu intervalo de linhas).

```
% smpicc -O3 1D_parallel_julia.c -o 1D_parallel_julia % smpirun -np 16
-hostfile ./simple_cluster_hostfile.txt -platform ./simple_cluster.xml
./1D_parallel_julia 100 [Process 1 out of 16]: I should compute pixel
rows 1 to 1, for a total of 1 rows [Process 2 out of 16]: I should
compute pixel rows 2 to 2, for a total of 1 rows [Process 3 out of
16]: I should compute pixel rows 3 to 3, for a total of 1 rows
[Process 4 out of 16]: I should compute pixel rows 4 to 4, for a total
of 1 rows [Process 5 out of 16]: I should compute pixel rows 5 to 5,
for a total of 1 rows [Process 6 out of 16]: I should compute pixel
rows 6 to 6, for a total of 1 rows [Process 7 out of 16]: I should
compute pixel rows 7 to 7, for a total of 1 rows [Process 8 out of
16]: I should compute pixel rows 8 to 8, for a total of 1 rows
[Process 9 out of 16]: I should compute pixel rows 9 to 9, for a total
of 1 rows [Process 10 out of 16]: I should compute pixel rows 10 to
10, for a total of 1 rows [Process 11 out of 16]: I should compute
pixel rows 11 to 11, for a total of 1 rows [Process 12 out of 16]: I
should compute pixel rows 12 to 11, for a total of 0 rows [Process 13
out of 16]: I should compute pixel rows 12 to 11, for a total of 0
rows [Process 14 out of 16]: I should compute pixel rows 12 to 11, for
a total of 0 rows [Process 15 out of 16]: I should compute pixel rows
12 to 11, for a total of 0 rows
```

Segunda Parte - Computação do Conjunto Usando MPI

Modifique o programa para que cada processo aloque espaço apenas para os pixels que precisa calcular e os processe. Como cada processo roda em um host diferente, essa abordagem permite calcular imagens grandes que não cabem na memória de um único host, característica essencial da computação distribuída.

Cada processo deve imprimir seu ranque e o tempo (usando `MPI_Wtime()`) imediatamente antes e depois do cálculo. Isso mostrará a diferença de desempenho entre os hosts, já que o cluster não é completamente homogêneo.

A maior dificuldade é manter claro o mapeamento entre os índices locais (do processo) e globais (da imagem). Embora a imagem seja vista como um array de n pixels, nenhum array com n pixels é alocado. Cada processo lida apenas com arrays menores, correspondentes à sua parte.

Terceira Parte - Salvando num Arquivo

Ajuste seu programa para salvar o conjunto de Julia em um arquivo BMP. Para isso, será necessário coordenar a execução entre os processos:

- Apenas um processo deve ser responsável por criar o arquivo e escrever o cabeçalho.
- Os processos devem gravar as linhas de pixels no arquivo em sequência, uma após a outra.
- Faça de tal forma que apenas um processo mantenha o arquivo aberto por vez.

O primeiro passo é escolher um processo para criar o arquivo e escrever o cabeçalho (normalmente, o processo de ranque 0). Depois disso, cada processo, ao terminar de calcular seus pixels, deve esperar por uma mensagem de "autorização" do processo anterior, gravar os dados no arquivo e, em seguida, enviar uma mensagem de "autorização" para o próximo processo. As exceções são: o processo de ranque 0, que não precisa esperar por uma mensagem, e o último processo, que não precisa enviar uma mensagem. O conteúdo dessas mensagens não é relevante (podem ser um único byte ou até mesmo mensagens vazias), pois o objetivo principal é sincronizar a ordem de gravação entre os processos, e não transferir dados significativos.

Atenção: Certifique-se de que os cálculos sejam realizados em paralelo. Ou seja, todos os processos devem calcular simultaneamente e, somente depois, se revezar para gravar os resultados no arquivo.

Teste seu programa com diferentes números de processadores e valores de n , garantindo que o arquivo BMP gerado seja válido e contenha a imagem correta.



Segunda Entrega (30%)

Código MPI funcional `1D_parallel_julia.c`. `Makefile` (opcional) que compile o alvo com SMPI.

Tarefa 3 - Experimentos

A última tarefa consiste em realizar alguns experimentos para analisar o desempenho e o comportamento do programa em diferentes cenários. Não é necessário testar todas as combinações possíveis de parâmetros sugeridos; **selecione-os de acordo com os objetivos de cada experimento.**

Experimentos com MPICH

Os primeiros testes devem ser executados localmente em sua máquina utilizando o MPICH. O objetivo é comparar o desempenho da versão paralela com a sequencial.

1. Medição de tempo e análise de desempenho:

- Meça e compare o tempo de execução para diferentes tamanhos de imagem (500, 1000, 5000) utilizando diferentes números de processos (2, 4, 8 e 16).

2. Speedup e eficiência:

- Calcule o speedup obtido pela versão paralela em relação à versão sequencial.
- Fixe o tamanho da imagem e aumente gradativamente o número de