Programação p Sistemas Paralelos e Distribuídos
Prof.: Fernando W. Cruz

Ordenação de vetores e manipulação de arquivos com MPI/OMP

Para as questões 1 a 4, considere o texto abaixo:

Os fractais são baseados no princípio de que um objeto geométrico pode ser dividido em partes menores, cada uma delas semelhante ao objeto original. São, portanto, objetos com muitos detalhes, com similaridade recursiva. Um dos fractais interessantes de se observar é o fractal Julia. Um conjunto Julia (*Julia set*) é uma generalização do famoso conjunto Mandelbrot [https://pt.wikipedia.org/wiki/Conjunto_de_Mandelbrot], que está ilustrado na Figura 1.

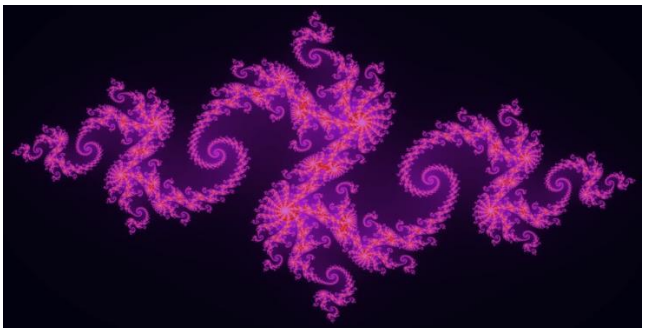

Figura 1 – Fractal Julia

Esse fractal é definido como segue. Dado `z` um ponto no plano complexo 2-D, calculamos a série definida como: $z_0 = z$ e $z_{n+1} = z_n^2 + c$, onde `c = -0,79 + i * 0,15`, ou seja, um número complexo. 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 seja superior a 2 (ou até que um número máximo de iterações seja atingido). O programa `fractal.c` é o código que produz a imagem da Figura 1, mas é possível alterá-lo para criar imagens diferentes. Este programa pode ser compilado com o parâmetro a seguir:

```
$ gcc fractal.c -o fractal -lm
```

Para executá-lo, basta digitar o comando

`$ ./fractal <N>`, onde N é a altura da figura do fractal (ou número de linhas). Esse parâmetro é utilizado para o cálculo da largura (2*N) e o cálculo da área do fractal (altura * largura * 3).

A saída desse programa é um arquivo em formato bmp (Bitmap) que pode ser aberto com qualquer editor de imagens do seu sistema operacional.

Pede-se o seguinte:

1) Escreva uma versão MPI do código `fractal.c`, com o nome `fractalmpiserial.c`, de modo que N seja dividido pelo número de processos criados e a gravação do arquivo aconteça serialmente e em ordem do menor *rank* para o maior. Por exemplo, se o programa foi chamado da seguinte forma:

   ```
   $ mpirun -np 4 fractalmpiserial 1000
   ```

Significa que a altura do fractal é 1000 linhas e cada *rank* (calculado em função do parâmetro np) irá gravar 250 linhas no arquivo de saída, de forma não simultânea (paralela), mas serial, na seguinte ordem: *(1º)* o *rank* 0 escreve as linhas de 0 a 249; *(2º)* o *rank* 1 escreve as linhas 250 a 499; *(3º)* o rank 2 escreve de 500 a 749 e, *(4º)* o rank 3 escreve as linhas de 750 a 999.

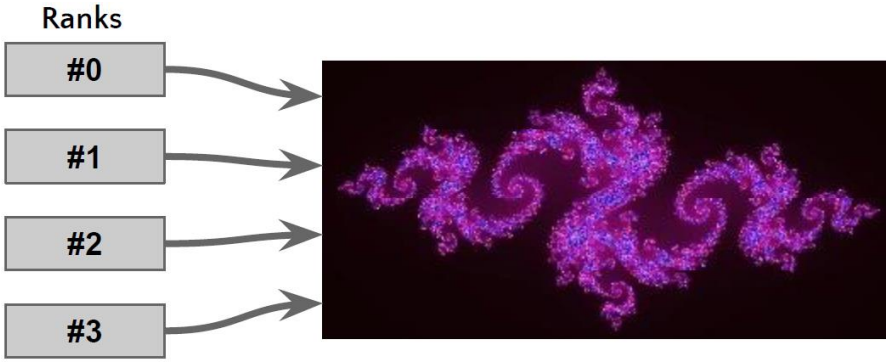

Figura 2 – Exemplo de execução MPI do Fractal Julia

Neste caso, a execução do código deve ser feita no cluster chococino, de modo que cada processo fique ativo em um *host* específico e cada um deles fique responsável por uma porção do arquivo de saída. Use um editor de imagens pra ver a figura resultante produzida.

2) Gerar uma segunda versão MPI do código fractal.c, com o nome `fractalmpi_io.c`, de modo que N seja dividido pelo número de processos, mas a gravação aconteça em paralelo, obedecendo o *offset* calculado para cada *rank*, e fazendo uso de alguma função de escrita de arquivo da biblioteca MPI. Conforme apresentado na Tabela 1, são muitas funções de I/O MPI, mas nem todas elas servem para o problema apresentado neste experimento. Portanto, a sugestão é fazer uma pesquisa, antes de adotar uma solução factível. Obs.: Pontos extras para as respostas que contemplarem mais de uma alternativa de função de gravação.

Tabela 1 – Funções MPI de I/O (Input/Output) em arquivos

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	blocking	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	nonblocking	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_iread_at_all MPI_File_iwrite_at_all
	split collective	N/A	MPI_File_read_at_all_begin/end MPI_File_write_at_all_begin/end
individual file pointers	blocking	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	nonblocking	MPI_File_iread MPI_File_iwrite	MPI_File_iread_all MPI_File_iwrite_all
	split collective	N/A	MPI_File_read_all_begin/end MPI_File_write_all_begin/end
shared file pointer	blocking	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	nonblocking	MPI_File_iread_shared MPI_File_iwrite_shared	N/A
	split collective	N/A	MPI_File_read_ordered_begin/end MPI_File_write_ordered_begin/end

Observação: Assim como na questão anterior, os processos devem ser disparados no cluster chococcino, cada processo em um *host* distinto e cada um deles responsável por uma porção do arquivo de saída. Ao final, é importante visualizar a figura resultante para ver se houve algum erro de processamento. Para isso, utilize um editor de imagens que consiga ler arquivos .bmp.

3) Gerar uma versão OpenMP do código `fractal.c`, com o nome `fractalomp.c`, de modo que N seja dividido pelo número de threads disparadas no momento da chamada. Nesse caso, cada thread deve ficar responsável por armazenar uma porção do arquivo de saída.

4) Para uma determinada dimensão do referido fractal, qual dos três programas montados apresenta melhor performance? Qual o percentual de ganho de uma solução em relação a outra? Para essa resposta, montar um experimento controlado, com simulações de execução dos códigos, levando-se em conta os parâmetros que influenciam a resposta (número de threads, de núcleos, de máquinas, etc.). Ao final dos testes, monte uma tabela comparativa e mostre os tempos de execução de cada programa, considerando os parâmetros que influenciam a performance e apresente uma resposta conclusiva.

5) A função `selection_sort()` da Figura 3, faz a ordenação serial de um vetor de inteiros. Proponha uma versão dessa função em OpenMP que tenha um tempo de resposta melhorado, pelo uso dos pragmas de paralelização. Para testar esse código, veja o programa `ordena_vetor.c` que está disponível junto com essa especificação.

```
1  void selection_sort(int *v, int n){
2      int i, j, min, tmp;
3
4      for(i = 0; i < n - 1; i++){
5          min = i;
6
7          for(j = i + 1; j < n; j++)
8              if(v[j] < v[min])
9                  min = j;
10
11         tmp = v[i];
12         v[i] = v[min];
13         v[min] = tmp;
14     }
15 }
```

Figura 3 – função de ordenação de vetores de inteiros