

Programação para Sistemas Paralelos e Distribuídos

Programação para Sistemas Paralelos e Distribuídos - Relatório

Nome: Bruno Henrique Sousa Duarte

Matrícula: 17/0138551

Este relatório é um compilado dos README.md presente em cada pasta das questões do repositório.

Questão 1

Este código é um programa em C que calcula a imagem de um fractal Julia em paralelo usando MPI (Message Passing Interface). O programa é dividido em vários processos que executam a computação em paralelo, e o resultado é escrito em um arquivo de imagem BMP.

O código começa definindo o nome do arquivo de saída e uma matriz que irá armazenar os intervalos de linhas que cada processo irá computar. A função `limits()` é responsável por dividir o trabalho entre os processos. Ela recebe o número de linhas da imagem e o número de processos e, em seguida, calcula os intervalos de linhas que cada processo deve computar.

A função `worker_write_pixel_lines()` é usada para escrever um conjunto de linhas de pixels em um arquivo BMP. Ela recebe o ponteiro para o arquivo de saída, as dimensões da imagem, a linha inicial, o número de linhas a serem escritas e o ponteiro para o array de pixels. A função usa a função `fseek()` para mover o ponteiro do arquivo para a posição correta e, em seguida, escreve as linhas de pixels usando a função `fwrite()`.

Em seguida, a função `main()` começa lendo o número de linhas da imagem como argumento de linha de comando e, em seguida, inicializa MPI. A função `gethostname()` é usada para obter o nome da máquina em que o processo está sendo executado.

Em seguida, o programa aloca memória para o array de pixels e para um array de três elementos usado para armazenar a cor de cada pixel. O programa, em seguida, executa um loop para cada intervalo de linhas atribuído a este processo. Para cada pixel, o programa chama a função `compute_julia_pixel()` (definida no arquivo `lib_julia.h`) para calcular sua cor e armazená-la no array de pixels.

O programa, em seguida, abre o arquivo de saída e chama a função `worker_write_pixel_lines()` para escrever as linhas de pixels calculadas por este processo no arquivo de saída. Finalmente, o programa desaloca a memória alocada e encerra a execução do MPI.

Questão 2

Na questão 2 temos um programa em C que usa MPI (Interface de Passagem de Mensagens) para calcular uma imagem da fractal de Julia e salvar a imagem em um arquivo BMP. O código é semelhante ao anterior (Questão 01), mas agora é dividido em intervalos para que vários processos possam calcular pixels diferentes e, em seguida, gravá-los no arquivo BMP usando `MPI_File_write_at()`.

Aqui estão as principais partes do código:

- inclui as bibliotecas necessárias, incluindo MPI. `int intervalos[1000][3]`: Uma matriz que armazena o intervalo de linhas que cada processo deve calcular.
- `limits()`: Uma função que calcula os limites de linha para cada processo com base no número total de linhas e no número de processos.
- `MPI_Init()`, `MPI_Comm_rank()`, `MPI_Comm_size()`, `MPI_Status`: Funções MPI para inicializar o MPI, obter o número de processos e o ID do processo, bem como definir o status da mensagem. `for (int i = intervalos[rank][0]; i < intervalos[rank][1]; i++)`: Um loop que itera pelas linhas que este processo deve calcular com base no seu ID de processo.
- `compute_julia_pixel()`: Uma função que calcula a cor de um pixel com base em sua posição na imagem e em alguns parâmetros de entrada. Essa função é definida em um arquivo de biblioteca separado chamado `lib_julia.h`.
- `MPI_File_write_at()`: Uma função MPI que grava o buffer de pixels em um arquivo BMP. Ele grava a partir de um deslocamento específico no arquivo para que cada processo grave na posição correta.
- `MPI_Finalize()`: Uma função MPI que finaliza o MPI.
- `MPI_File_write_at` vs `MPI_File_write` (Extra). O `MPI_File_write_at` é semelhante ao `MPI_File_write`, mas permite que o processo escreva em um deslocamento específico no arquivo, em vez de começar a escrever no início do arquivo. Isso pode ser útil em situações em que os processos precisam escrever em diferentes locais do arquivo, sem precisar coordenar essas operações.

Em geral, o programa divide o cálculo de pixels em várias partes para que diferentes processos possam trabalhar em diferentes partes da imagem. Cada



processo calcula seus pixels e, em seguida, grava-os em um arquivo BMP usando `MPI_File_write_at()`.

Questão 3

Este é um programa em linguagem C que realiza a ordenação de um vetor de inteiros usando o algoritmo selection sort e o algoritmo qsort da biblioteca padrão da linguagem.

Primeiro ocorre a inclusão das bibliotecas `stdio.h`, `stdlib.h`, `string.h`, `time.h` e `omp.h`. A Declaração da função `selection_sort`, que recebe um ponteiro para um vetor de inteiros e o número de elementos do vetor, e realiza a ordenação do vetor usando o algoritmo selection sort e de semelhante forma a declaração da função `ompsort`, que recebe um ponteiro para um vetor de inteiros, um ponteiro para um vetor de saída e o número de elementos do vetor, e realiza a ordenação do vetor usando o algoritmo selection sort em paralelo usando OpenMP. Aloca memória para os vetores de entrada (`vector`) e de saída (`out`), ambos com tamanho `n` e a local memória para um vetor de teste (`test`) com tamanho `n`. Preenche o vetor `vector` com números inteiros aleatórios usando a função `rand()`. Copia o conteúdo do vetor `vector` para o vetor `test`. Chama a função `qsort` para ordenar o vetor `test` em ordem crescente e mede o tempo de execução. Chama a função `ompsort` para ordenar o vetor `vector` em ordem crescente usando OpenMP e mede o tempo de execução. Exibe os primeiros 20 elementos do vetor `test` e do vetor `out` para verificar se a ordenação foi feita corretamente e compara o conteúdo dos vetores `test` e `out` para verificar se a ordenação foi feita corretamente. Exibe uma mensagem indicando se a ordenação foi feita corretamente e os tempos de execução do `qsort` e do `ompsort`, e por fim, libera a memória alocada para os vetores.



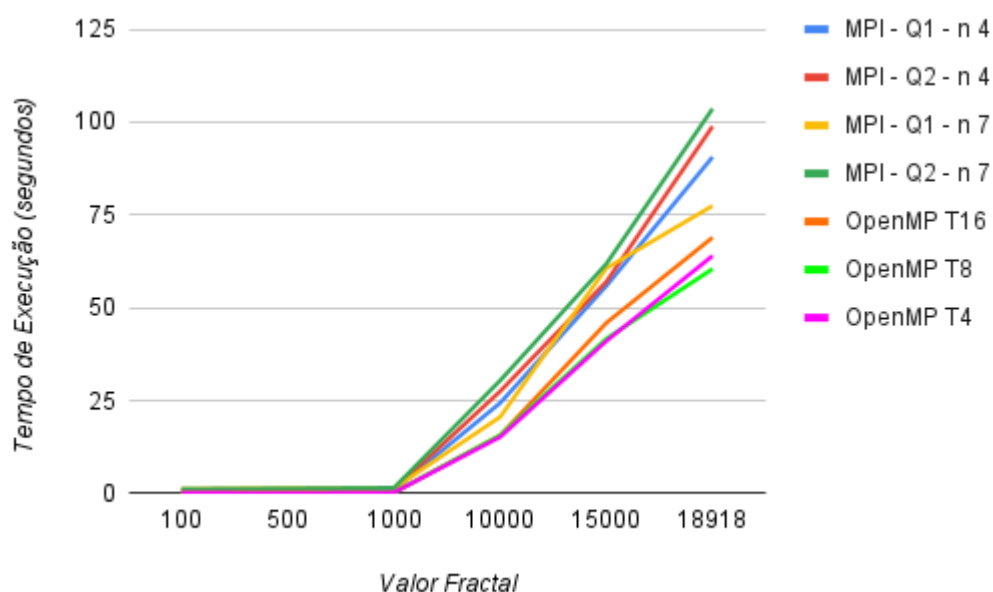
Questão 4) Para uma determinada dimensão do referido fractal, qual dos três programas montados apresenta melhor performance? Qual o percentual de ganho de uma solução em relação a outra? Para essa resposta, montar um experimento controlado, com simulações de execução dos códigos, levando-se em conta 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.

Fractal	100	500	1000	10000	15000	18918
MPI - Questão 1 -n 4	0.724s	0.825s	0.962s	24.370s	55.922s	1m30.632s
MPI - Questão 2 -n 4	0.923s	0.829s	1.076s	27.520s	57.021s	1m38.878s
MPI - Questão 1 -n 7	1.195s	1.388s	1.421s	20.598s	1m0.662s	1m17.485s
MPI - Questão 2 -n 7	1.193s	1.337s	1.531s	30.419s	1m1.853s	1m43.642s
OpenMP \$OMP_NUM_THRE ADS = 16	0.016s	0m0.157s	0.249s	15.648s	45.942s	1m8.951s
OpenMP \$OMP_NUM_THRE ADS = 8	0.020s	0.132s	0.239s	15.660s	41.699s	1m0.472s
OpenMP \$OMP_NUM_THRE ADS = 4	0m0.205 s	0m0.102s	0m0.232s	15.213s	41.065s	1m4.008s

A partir da tabela podemos concluir que o **MPI** (Message Passing Interface) que permite vários processos em diferentes hosts se comunicam entre si e coordenam seu trabalho em uma tarefa paralela. Quando se utiliza MPI em mais de um host, o tempo de execução pode ser afetado por diversos fatores, como a latência e a largura de banda da rede, a carga de trabalho distribuída entre os hosts e a eficiência do algoritmo paralelo implementado. Assim, conforme podemos observar o tempo de execução para 7 hosts seja superior ao tempo de execução para 4 hosts. Isso pode ocorrer devido a vários motivos, como: **Sobrecarga da rede**, quanto mais hosts são utilizados, maior é o tráfego de dados na rede e maior é a latência de comunicação entre os hosts. Isso pode levar a atrasos na troca de mensagens entre os processos MPI e, conseqüentemente, aumentar o tempo de execução e **Balanceamento de carga**, ao utilizar mais hosts, é necessário distribuir a carga de trabalho entre eles de forma equilibrada. Se essa distribuição não for eficiente, alguns hosts podem ficar ociosos enquanto outros estão sobrecarregados, o que pode diminuir a eficiência do algoritmo paralelo e aumentar o tempo de execução.

Em relação ao **OpenMP** é a capacidade de executar tarefas em paralelo por meio do uso de threads. Quando se utiliza OpenMP, aumentar o número de threads pode melhorar o desempenho em certas condições. Isso ocorre porque, em geral, o aumento do número de threads leva a uma distribuição mais equilibrada da carga de trabalho entre os processadores e reduz o tempo de espera ocioso para acesso à memória compartilhada.

Portanto, conforme observado na tabela, o OpenMP permite que essas partes do código sejam executadas em paralelo por meio do uso de threads, aproveitando a memória compartilhada disponível no sistema e alcançando uma escalabilidade muito boa e um desempenho alto. Por outro lado, o MPI é mais adequado para aplicações que requerem comunicação e coordenação entre múltiplos processos em sistemas distribuídos. O MPI permite que os processos se comuniquem e coordenem entre si, permitindo que eles realizem cálculos paralelos em vários sistemas. O limite dos valores $[100, 18918]$ de Fractais se baseia até onde o chococino calculava o fractal sem dar falha de segmentação.





Exemplos de Execução: Os tempos da tabela acima foram obtidos mediante chococino.

Questão 1)

Local:

```
gcc -c lib_julia.c
mpicc lib_julia.o fractalmpiserial.c -o fractalmpiserial -lm
mpirun -n 8 ./fractalmpiserial 1000
```

ou

Chococcino:

```
gcc -c lib_julia.c
mpicc lib_julia.o fractalmpiserial.c -o fractalmpiserial -lm
mpirun -host cm1,cm2,cm3,cm4,gpu1,gpu2,gpu3 -n 7 ./fractalmpiserial 1000
```

ou

make

```
mpirun -host cm1,cm2,cm3,cm4,gpu1,gpu2,gpu3 -n 7 ./fractalmpiserial 1000
```

Questão 2)

```
gcc -c lib_julia.c
mpicc lib_julia.o fractalmpi_io.c -o fractalmpi_io -lm
mpirun -n 4 ./fractalmpi_io 1000
```

ou

```
gcc -c lib_julia.c
mpicc lib_julia.o fractalmpi_io.c -o fractalmpi_io -lm
mpirun -host cm1,cm2,cm3,cm4 -n 4 ./fractalmpi_io 1000
```

ou

make

```
mpirun -host cm1,cm2,cm3,cm4,gpu1,gpu2,gpu3 -n 7 ./fractalmpi_io 1000
```

Questão 3)

make

```
make run V_THREADS=8 N_FRACTAL=1000
```



Questão 5

Este é um programa em linguagem C que realiza a ordenação de um vetor de inteiros usando o algoritmo selection sort e o algoritmo qsort da biblioteca padrão da linguagem.

Primeiro ocorre a inclusão das bibliotecas *stdio.h*, *stdlib.h*, *string.h*, *time.h* e *omp.h*. A Declaração da função *selection_sort*, que recebe um ponteiro para um vetor de inteiros e o número de elementos do vetor, e realiza a ordenação do vetor usando o algoritmo selection sort e de semelhante forma a declaração da função *ompsort*, que recebe um ponteiro para um vetor de inteiros, um ponteiro para um vetor de saída e o número de elementos do vetor, e realiza a ordenação do vetor usando o algoritmo selection sort em paralelo usando OpenMP. Aloca memória para os vetores de entrada (*vector*) e de saída (*out*), ambos com tamanho n e a local memória para um vetor de teste (*test*) com tamanho n . Preenche o vetor *vector* com números inteiros aleatórios usando a função *rand()*. Copia o conteúdo do vetor *vector* para o vetor *test*. Chama a função *qsort* para ordenar o vetor *test* em ordem crescente e mede o tempo de execução. Chama a função *ompsort* para ordenar o vetor *vector* em ordem crescente usando OpenMP e mede o tempo de execução. Exibe os primeiros 20 elementos do vetor *test* e do vetor *out* para verificar se a ordenação foi feita corretamente e compara o conteúdo dos vetores *test* e *out* para verificar se a ordenação foi feita corretamente. Exibe uma mensagem indicando se a ordenação foi feita corretamente e os tempos de execução do *qsort* e do *ompsort*, e por fim, libera a memória alocada para os vetores.