Trabalho de Fundamentos de Sistemas Paralelos e Distribuídos

Paralelizando cálculos do conjunto de Mandelbrot com Pthreads

Daniel Souza de Campos

Ciência da Computação – UFMG – 2021/2

Sumário

1 – Introdução

O Conjunto de Mandelbrot é constituído pelos números para qual uma expressão específica, se executada iterativamente, converge. O interessante é que conjunto dos números para qual a equação converge pode ser visto como um fractal ao levar em consideração o número de iterações necessárias para detectar a convergência.

É possível conferir a natureza fractal do Conjunto de Mandelbrot ao visualizar o próprio conjunto de acordo com intervalos de números reais e complexos de interesse. Se esse intervalo estiver contido em uma região onde são necessárias muitas iterações para verificar se um ponto pertence ao conjunto ou não, o custo computacional e de tempo se torna grande. Além disso, realizar a computação usando apenas uma Thread desperdiça a capacidade das máquinas modernas.

Para paralelizar esse processo, podemos dividir a região dada pelo intervalo fornecido em regiões menores. Dessa forma, já que o cálculo para um ponto não depende de outros pontos, pode-se fazer com que cada thread trabalhe paralelamente às outras focando em sua própria região de cálculo.

O objetivo desse trabalho é aplicar a abordagem de paralelização acima no cálculo de regiões para verificação do Conjunto de Mandelbrot utilizando Pthreads.

2 – Ferramentas utilizadas

O trabalho foi desenvolvido utilizando a linguagem C++11 e compilador g++ no editor de códigos *Visual Studio Code*. A máquina de desenvolvimento é um notebook Dell com 8GB RAM e um processador Intel core I5 de 7º geração. A máquina possui o sistema operacional Windows 10 com o *Windows Subsystem for Linux* (WSL) instalado, o que permitiu o desenvolvimento e execução do programa compilado com a biblioteca Pthreads.

O Github foi utilizado para controle de versionamento e como repositório do código e estará disponível publicamente em https://github.com/Pendulun/ParallelMandelbrot após a data final de entrega do trabalho dia 03/01/2022.

3 – Compilação e execução

O projeto possui uma pasta chamada *ex1* dentro da pasta raiz. Dentro dessa pasta está o único arquivo com todo o código de execução e um arquivo *Makefile*. Para compilar o programa, basta executar *make* ou *make build* na pasta *ex1* que um arquivo executável de nome *prog* será gerado.

Para executar o programa pode-se fazê-lo diretamente via "*./prog ../mandelbrot_tasks/algumArquivo <número de Threads trabalhadoras>"* ou via o make com o comando '*make run ARGS="../mandelbrot_tasks/algumArquivo <número de Threads trabalhadoras>" '*.

O número de Threads trabalhadoras é opcional e terá o valor padrão de 4. Se for informado, o programa assume que ele será um número inteiro maior ou igual a 1.

4 – Código original

Foi disponibilizado, de antemão, um código na linguagem de programação C que já possuía a capacidade de ler um arquivo de entrada com regiões onde o cálculo para o Conjunto de Mandelbrot deveria ser realizado. Além disso, o código para o próprio cálculo também já havia sido disponibilizado.

Dessa forma, o trabalho deve modificar o código disponibilizado para que várias Threads trabalhem simultaneamente a fim de agilizar o cálculo em todas as regiões especificadas no arquivo de entrada.

O código original é dividido em duas funções principais. A primeira lê uma linha do arquivo de entrada e preenche uma struct com as informações da região onde o cálculo deve ser realizado. A segunda parte recebe essa struct e realiza, de fato, o cálculo. Quando esse cálculo acabar, mais uma linha era lida e acontecia o cálculo novamente até que todas as linhas tivessem sido lidas.

5 – Modificações no código original

Como sugerido na especificação do trabalho, deve ser usado uma espécie de fila de onde as várias threads podem consumir structs com atributos das regiões especificadas para o cálculo necessário. Dessa forma, será necessária uma coordenação do acesso à fila por parte das Threads para que nenhuma condição de corrida possa afetar o desempenho do programa.

Também foi especificado a existência de dois tipos de Threads:

1. Uma única Thread de leitura do arquivo de entrada que preenche a fila de structs de regiões
2. Threads de consumo da fila de structs e que realizam o cálculo já pronto para a região conhecidas como Threads trabalhadoras.

Uma das restrições do trabalho era que a fila deve conter um número de structs de, no máximo, 4 vezes o número de Threads trabalhadoras. Além disso, quando uma Thread trabalhadora percebe que a fila contém um número de structs igual ou menor do que o número de Threads trabalhadoras, ela deve notificar a Thread de leitura para que ela preencha a fila.

Assim, pode-se notar uma característica da Thread de leitura: ela só preenche a fila quando é notificada pelas Threads trabalhadoras e permanece dormindo enquanto não for acordada.

Dessa forma, pode-se estabelecer uma comparação entre o contexto do problema e um problema menos clássico de sincronização, como dito no livro *Little Book Of Semaphores*, o Jantar dos Selvagens.

Nesse problema, existe um conjunto de selvagens que ficam consumindo comida de uma panela. Nenhum selvagem pode pegar da panela ao mesmo tempo. Quando um selvagem percebe que a panela atingiu um nível mínimo de comida, ele chama um garçom que pega a panela, a preenche com comida e a devolve para a mesa. Nesse meio tempo, nenhum selvagem pode pegar comida da panela.

Desse modo, pode-se modelar o problema fazendo com que a Thread de Leitura seja o garçom, as Threads trabalhadoras sejam os selvagens, a panela seja a fila de structs, a comida seja as structs em si e o ato de comer seja realizar o cálculo a partir da struct de regiões.

O código da solução do trabalho foi baseado no código de exemplo da seção sobre o Jantar Dos Selvagens do *Little Book Of Semaphores* (Figura 1) e modificado para o uso de Pthreads para atender o contexto do trabalho.

5.1 – Fluxo de execução

Na sua própria função, a Thread de leitura, primeiramente, espera um sinal de alguma Thread trabalhadora para preencher a fila. Isso é feito utilizando uma variável de condição chamada *empty_queue* associada a um *mutex empty_queue_mutex* de acordo com a necessidade de Pthreads.

Uma Thread trabalhadora tenta acesso a fila via um *mutex* específico para acesso à fila das trabalhadoras chamado *workers_queue_access*. Uma vez conseguido esse acesso, ela verifica se a fila contém a quantidade mínima de structs. Se não, ela sinaliza a variável de condição *empty_queue*, acordando a Thread de leitura.

Dining Savages solution (cook)

```
while True:
    emptyPot.wait()
    putServingsInPot(M)
    fullPot.signal()
```

Dining Savages solution (savage)

```
while True:
    mutex.wait()
        if servings == 0:
            emptyPot.signal()
            fullPot.wait()
            servings = M
        servings -= 1
        getServingFromPot()
    mutex.signal()

    eat()
```

Figura 1: Código solução do Jantar Dos Selvagens no livro Little Book of Semaphores

Depois disso, a própria Thread trabalhadora espera a Thread de leitura indicar que ela preencheu a fila por meio de outra variável de condição chamada *filled_queue* associada a um *mutex filled_queue_mutex*.

Uma vez acordada, a Thread de leitura lê o arquivo de entrada linha por linha criando instâncias da struct que contém informações das regiões a serem computadas e as adiciona a fila sem passar do limite estabelecido. Se a Thread de leitura chegar ao fim do arquivo, ela produz structs de regiões com características específicas que indicam para a Thread que pegar essa struct que o trabalho já acabou. Essas structs são conhecidas como structs *End-Of-Work*

(EOW). Essas structs EOW são inseridas na fila e isso pode passar do tamanho máximo estabelecido. Isso foi implementado dessa forma assumindo que a restrição de tamanho da fila valia apenas para structs que realmente representassem trabalho a ser realizado.

Independente se a Thread de leitura chegou no final do arquivo, ela sinaliza para a variável de condição *filled_queue* que ela preencheu a fila. Importante notar que apenas uma Thread trabalhadora estará esperando por esse sinal já que apenas ela conseguiu a trava do *mutex workers_queue_access*. Depois, se ela não atingiu o final do arquivo, ela volta a esperar por um sinal da variável de condição *empty_queue*. Se chegou ao final do arquivo, ela simplesmente sai do seu loop principal de execução.

Uma vez acordada, a Thread trabalhadora pode ler da fila de structs. Ao pegar uma struct, ela verifica se ela não é uma struct EOW. Se for, ela sabe que o seu trabalho acabou. Além disso, ela sinaliza para as outras Threads que uma struct EOW foi identificada pela variável global booleana *startedEOWs*.

Após pegar uma struct, é feito novamente a verificação do tamanho mínimo da fila como descrito anteriormente se a struct não for EOW. Isso foi feito de acordo com a especificação do trabalho.

A variável *startedEOWs* será levada em consideração na verificação do tamanho mínimo da fila para as próximas Threads já que, se começaram a aparecer as structs EOW, o tamanho da fila não deve ser verificado e apenas devemos consumir dela. No final, a Thread trabalhadora libera o acesso à fila para as outras Threads e, se a struct não for EOW, ela começa o cálculo da região via a função *fractal*.

Durante todo esse processo, algumas estatísticas são coletadas para serem mostradas no final para o usuário.

6 – Testes realizados e resultados

De acordo com a especificação do trabalho, deveriam ser realizados testes do desempenho do programa levando em consideração dois valores para a quantidade de Threads trabalhadoras: uma e o máximo representado pelo número de núcleos na máquina de execução (que, no caso, é 4).

Foi utilizado o arquivo *mandelbrot_tasks/t* que contém especificações de 64 regiões para cálculo.

Foram realizadas 15 execuções para cada número de Threads trabalhadoras e coletadas algumas estatísticas sobre cada iteração.

Estatísticas coletadas para 1 Thread trabalhadora			
Iteração	Tempo médio tarefa (ms)	Desvio Tempo médio tarefa (ms)	Total Exec Time (ms)
1	208.4	409.8	13338
2	209.09	414.48	13382
3	212.25	422.44	13584
4	206.45	408.83	13213
5	207.5	410.78	13280
6	209.64	416.9	13417
7	206.21	408.75	13198
8	228.71	448.1	14638
9	210.48	416.87	13471
10	208.96	413.16	13374
11	206.62	409.78	13224
12	209.78	416.06	13426
13	208.76	412.15	13361
14	209.06	414.91	13380
15	210.76	417.58	13489
MÉDIA	210.178	416.0393333	13451.66667
DESVIO	5.21004952	9.347591823	333.49656

Tabela 1: Estatísticas de execução para 1 Thread trabalhadora

Estatísticas coletadas para 4 Threads trabalhadoras			
Iteração	Tempo médio tarefa (ms)	Desvio Tempo médio tarefa (ms)	Max Total Exec Time (ms)
1	477.68	963.15	8100.00
2	434.92	866.48	7346.00
3	401.92	802.04	6896.00
4	364.15	721.80	6361.00
5	483.45	972.39	8167.00
6	334.28	667.33	5852.00
7	409.32	808.88	6917.00
8	460.29	920.14	7857.00
9	393.17	793.90	6733.00
10	330.82	647.44	5640.00
11	341.03	675.50	5873.00
12	396.57	794.67	6802.00
13	392.40	770.97	6792.00
14	357.64	704.79	6169.00
15	386.42	762.61	6488.00
MÉDIA	397.60	791.47	6799.53
DESVIO	47.48	99.35	765.65

Tabela 2: Estatísticas de execução para 4 Threads Trabalhadoras

Num Threads	Tempo médio tarefa (ms)		Desvio Tempo médio tarefa (ms)		Max total Exec Time (ms)	
	MÉDIA	DESVIO P.	MÉDIA	DESVIO P.	MÉDIA	DESVIO P.
1	210.18	5.21	416.04	9.35	13451.67	333.50
4	397.60	47.48	791.47	99.35	6799.53	765.65

Tabela 3: Comparação das estatísticas para as execuções com Threads diferentes

De acordo com os resultados mostrados, pode-se perceber que, ao utilizar apenas uma Thread trabalhadora, o tempo médio de execução das tarefas foi de cerca de 210 milissegundos, enquanto que, ao utilizar 4 Threads trabalhadoras, esse tempo de execução subiu para cerca de 397 milissegundos por tarefa. Esse resultado não faz muito sentido se pensarmos que esse tempo de execução é cronometrado apenas com relação à função *fractal* comentada anteriormente.

De qualquer forma, como são utilizadas várias Threads, cada uma realizando cálculos de uma região diferente, o tempo máximo de execução por Thread diminuiu de cerca de 13451 milissegundos (13,451 segundos) com uma Thread para cerca de 6799 milissegundos (6,799 segundos) com 4 Threads.

Pensando que, teoricamente, as 64 regiões foram divididas em 4 Threads, mas que o tempo máximo de execução caiu por cerca da metade, pode-se inferir que alguma Thread realizou um ou mais cálculos mais custosos enquanto as outras realizaram várias outros cálculos menores. Isso é baseado também na Tabela 4 que mostra o desvio padrão médio com relação às 16 tarefas que seriam alocadas para cada Thread.

Desvio da média (16) de tarefas por Thread (4 Threads)	
Iteração	Desvio
1	8.28
2	7.61
3	7.52
4	8.28
5	6.05
6	6.92
7	8.12
8	8.28
9	3.65
10	6.78
11	6.37
12	6.05
13	7.61
14	5.59
15	9.30
MÉDIA	7.09
DESVIO	1.36

Tabela 4: Desvio padrão médio da quantidade de tarefas alocadas para cada uma das 4 Threads. Assume que a média era 16.

Isso pode indicar um desbalanceamento de carga que não foi verificado e estimado levando em consideração o tamanho da região a ser calculada para uma Thread. Para resolver isso, seria necessária outra abordagem para alocar regiões a Threads, podendo ter que deixar de ser da forma em que uma Thread ativamente pede por uma região para o modo em que a Thread trabalhadora fica esperando ser alocada a uma região escolhida pela Thread de leitura sendo que essa última dividiria as regiões de acordo com estimativas do custo do trabalho a ser realizado.

7 – Conclusão

O trabalho tinha como objetivo implementar uma paralelização do código original fornecido em Pthreads. Dessa forma, várias Threads trabalhadoras conseguem executar a mesma função de cálculo para entradas diferentes em paralelo visto que um cálculo não depende do outro. Assim, consegue-se aproveitar melhor a capacidade dos computadores modernos para agilizar o processamento de tarefas custosas, repetidas e independentes.

O objetivo do trabalho foi alcançado já que, com a modelagem do problema a partir de uma situação bem conhecida, o Jantar dos Selvagens, foi possível implementar uma solução adequada e satisfatória.

Os resultados dos testes mostraram que a execução para o arquivo de entrada melhorou em cerca de duas vezes ao quadruplicar o número de Threads trabalhadoras. Entretanto, ao levar em consideração a quantidade de tarefas realizadas por cada Thread, pode-se inferir que ocorre desbalanceamento de carga já que são as próprias Threads trabalhadoras que buscam a instância para computarem na fila. Tal fila poderia ser ordenada por um custo estimado de computação para cada região e usada para melhor distribuir a carga entre as Threads trabalhadoras.

Para finalizar, com esse trabalho foi possível por em prática os conceitos estudados em aula, especificamente Pthreads e um problema clássico de sincronização. Eles serviram para mostrar a capacidade do computador de agilizar a execução de tarefas quando o programador leva em consideração as múltiplas threads disponíveis no projeto do software.

8 – Referências

An Introduction To Parallel Proggraming; Pacheco, Peter; Elselvier*; 2011*

The Little Book of Semaphores; Downey*,* Alley*; 2016;* Versão *2.2.1;* Disponível em https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf. Último acesso em 02/01/2022.