Exercício de Programação 1
Mandelbrot paralelizado

Arthur Souto Lima

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

arthursl@ufmg.br

1. Introdução

Na geometria fractal, o conjunto de Mandelbrot é um conjunto de números complexos para os quais uma determinada função complexa converge [Wikipedia 2021]. Graficamente, representações do conjunto de Mandelbrot criam imagens fractais fascinantes, principalmente por apresentar detalhes recursivos cada vez mais finos. Para gerar tais representações, são necessárias várias iterações a fim de detectar a divergência ou não da função naquele ponto específico. Essa computação, principalmente por ser independente para cada ponto, permite uma paralelização, o que melhora o desempenho para a geração de tais imagens. O objetivo desse exercício é gerenciar a criação da representação do conjunto de Mandelbrot por meio de uma fila de tarefas que é consumida por threads trabalhadoras, seguindo o modelo do Jantar dos Selvagens, como visto em [Maziero 2019].

2. Decisões de Projeto

A implementação da semântica de sincronização entre as tarefas segue a solução do Jantar dos Selvagens vista em [Maziero 2019]. Contudo, como requisitado pela especificação, ela foi adaptada para usar apenas variáveis de condição e mutexes.

A fim de facilitar a implementação, o programa foi modularizado em arquivos diferentes. Assim, o código do cozinheiro está no módulo *cook.h*, o dos trabalhadores no *worker.h*, o para cômputo dos fractais em *mandel-tiles-graphic.h* (adaptado do que foi fornecido) e auxiliares de leitura e processamento de arquivo em *parsing.h*. Os módulos estão na pasta *src* e os headers na pasta *include*. O código principal, junto com a função *main*, se encontra no arquivo *main.cpp* na pasta raiz do programa. É incluso um makefile para compilação e execução do programa, tal como orientado pela especificação.

Além deste breve relatório, o leitor é convidado a averiguar o código, no qual, na medida do possível, intentou-se comentar e documentar as diversas decisões e ações tomadas a cada passo. Dentre os procedimentos escritos, o maior e mais complexo é a thread do cozinheiro *cook_thread*, do módulo *cook.h*, pois ele é a "alma" de todo o programa. É ele quem crias threads, suas estruturas auxiliares, computa as estatísticas e faz o preenchimento da fila.

2.1. Execução

Como orientado pelas orientações, o programa deve receber dois argumentos para execução, nessa ordem, o nome ou caminho para o arquivo de entrada, obrigatório, e a quantidade de threads trabalhadoras a serem criadas, opcional.

Para o parsing desses argumentos, adaptou-se a estrutura que foi fornecida, a qual lia uma paleta de cores opcional e inteira como segundo argumento. Foi uma alteração bem simples de se implementar.

Assim, usando o *make run*, são chamadas válidas para a execução do programa:

```
$ make run ARGS="mandelbrot_tasks/t 6"
$ make run ARGS="mandelbrot_tasks/t"
```

Sendo que a primeira executa o programa usando como entrada o arquivo *t*, que está dentro da pasta *mandelbrot_tasks* e usará 6 threads trabalhadoras. O segundo também usa esse mesmo arquivo, mas usará o padrão de 4 threads trabalhadoras. Esse padrão está definido numa constante no arquivo *main.cpp*.

Nos dois casos, o tamanho da fila será de 4 vezes a quantidade de threads. A única forma de alterar esse comportamento é alterando a constante que controla o fator $\frac{\text{Tamanho da Fila}}{\text{Quantidade de Threads}}$, atualmente em 4, definida também no arquivo *main.cpp*.

2.2. Fila de Tarefas

A fila de tarefas é um *deque* da STL de C++, ou seja, suporta inserção e remoção segundo o paradigma FIFO, próprio das filas, mas ainda permite acesso aos elementos guardados sem necessariamente retirá-los, como um *vector*. Essa última característica é muito útil para testes, por isso a escolha por essa estrutura ao invés da *queue* da STL.

Essa fila guarda ponteiros para *structs fractal_param_t*, em que cada uma representa uma tarefa. A única seção do código que é capaz de preencher a fila é a thread do cozinheiro (em *cook.h*) e a única capaz de retirar tarefas são as threads dos trabalhadores (em *worker.h*).

A ideia geral da evolução da fila e do programa é a seguinte:

1. Cozinheiro preenche a fila com tarefas inicialmente
2. Cozinheiro cria as threads trabalhadoras e aguarda na variável de condição *cook_needed*
3. Threads requisitam acesso à fila via mutex
4. Uma consegue, retira uma tarefa da fila, libera o mutex
5. Threads retiram tarefas dessa forma
6. Fila fica vazia e (uma ou mais) threads acordam o cozinheiro
7. Cozinheiro preenche novamente com mais tarefas e/ou com EOW e volta a aguardar na variável de condição
8. Threads trabalhadoras vão retirando EOW e terminando
9. Cozinheiro preenche mais EOW, se for o caso, após a fila ficar vazia novamente e ele ser acordado
10. Threads trabalhadoras terminam
11. Cozinheiro espera as threads trabalhadoras terminares
12. Cozinheiro computa as estatísticas e termina

Para acesso à fila, como já ressaltado, tanto para preenchê-la quanto para retirar uma tarefa, é necessária posse da mutex que faz esse controle. As duas variáveis de condição servem para auxiliar o processo: a primeira, *cook_needed*, para que o cozinheiro possa ser suspenso até ser requisitado para preencher novamente a fila e a outra, *pot_filled*,

para avisar aos trabalhadores, que foram suspensos quando a fila se esvaziou, que ela já está cheia novamente.

Segundo a biblioteca *pthreads*, usada para implementar essa sincronização, ao realizar um *wait* numa variável de condição, o mutex passado é liberado. Assim, o cozinheiro ao fazer o *wait* libera o mutex de acesso à fila para que alguma thread trabalhadora acesse a fila. Da mesma forma, ao ser acordada, a thread recebe novamente o mutex.

Outro detalhe é que como só há um cozinheiro, então basta um *signal* na variável *cook_needed* para acordá-lo. Contudo, como potencialmente mais de um trabalhador esperando na variável *pot_filled*, há de se usar um *broadcast* para acordá-los e, devido ao *while*, apenas um conseguirá acesso ao mutex da fila.

2.3. Coleta de Estatísticas

A tarefa de coletar estatísticas parecia desafiadora no início da implementação, potencialmente exigindo mais estruturas de sincronização. Contudo isso não foi necessário, por alguns motivos.

Em primeiro lugar, como já comentado anteriormente, somente uma estrutura preenche a fila e somente outra remove elementos, logo podemos colocar contadores locais no cozinheiro para somar quantas tarefas foram colocadas na fila. Da mesma forma, pode-se colocar contadores nos trabalhadores para que cada um conte quantas tarefas pegou da fila e computou.

Em segundo lugar, o cálculo dessas estatísticas era feito somente ao final da thread do cozinheiro, ou seja, quando todas as threads trabalhadoras já haviam terminado. Assim, não havia mais potencial situação de concorrência que exigisse sincronização desses dados.

O único detalhe é como esses dados coletados pelo trabalhador voltavam para o cozinheiro. Isso era feito através da estrutura passada para cada thread trabalhadora. Ela continha os argumentos, como o ponteiro para acesso à fila, mas também alguns valores de ”retorno”, onde seriam colocados quantas tarefas ele fez e os respectivos tempos (guardados num *vector* de *doubles*) delas. Ao final, após as threads terminarem, o cozinheiro, que é quem criou as threads trabalhadoras e também quem criou essas *structs* de comunicação, pode acessá-las novamente para coletar as estatísticas e computar os dados requeridos pela especificação.

3. Análise Experimental

3.1. Metodologia

Serão feitos alguns testes para avaliação do desempenho do programa em alguns parâmetros como tempo médio por tarefa, quantidade de tarefas por thread e quantas vezes as threads trabalhadoras encontraram a fila vazia. A quantidade de threads trabalhadoras a serem criadas e também o tamanho da fila são definidos por duas constantes no arquivo *main.cpp*.

A máquina utilizada nesses testes é uma munida de um processador de 6 núcleos, executando em ambiente Windows na plataforma WSL (Windows Subsystem for Linux) Ubuntu 20.04. Compilaremos o programa usando o compilador GNU G++ da GNU Compiler Collection, em sua versão 9.3.0, de 2019.

Serão quatro cenários de teste. Os dois primeiros tem como entrada o arquivo *t* fornecido juntamente com o material deste exercício prático. Os dois últimos usam como entrada um arquivo *z3* criado para testes, com 16 tarefas, sendo 14 mais rápidas e 2 lentas, ou seja, propositalmente desbalanceado. Este arquivo também é fornecido juntamente com os arquivos entregues.

Cada um dos arquivos será usado para um teste em que o tamanho da fila é exatamente igual à quantidade de threads. O outro teste ao qual os dois arquivos serão submetidos é um em que o tamanho da fila é exatamente 4 vezes o número de threads. Esse fator, segundo a especificação, é o tamanho máximo da fila e está definido por uma constante no código.

3.2. Resultados e Discussão

Os resultados do arquivo *t* podem ser observados nas tabelas 1 e 2, enquanto os do arquivo *z3* estão nas tabelas 3 e 4.

Tabela 1. Resultados do arquivo *t*, fila igual ao número de threads

Qtd. threads	Tamanho Fila	Tempo por tarefa (média) (ms)	Desvio do tempo por tarefa (ms)	Média por trabalhador	Desvio da média por trabalhador	Qtd. vezes de fila vazia
1	1	149,782750	292,337144	64	0	64
2	2	365,553312	723,700611	32	1	33
3	3	600,541172	1.178,027659	21,333333	10,338708	22
4	4	911,171953	1.776,126831	16	6,204837	18
6	6	1.395,348344	2.740,768728	10,666667	3,771236	13

Tabela 2. Resultados do arquivo *t*, fila igual a 4 vezes o número de threads

Qtd. threads	Tamanho Fila	Tempo por tarefa (média) (ms)	Desvio do tempo por tarefa (ms)	Média por trabalhador	Desvio da média por trabalhador	Qtd. vezes de fila vazia
1	4	149,802703	293,029400	64	0	16
2	8	358,161594	710,802664	32	2	9
3	12	668,577422	1.316,493563	21,333333	10,077478	6
4	16	813,872031	1.578,740244	16	5,431390	6
6	24	1.143,090531	2.393,809778	10,666667	5,587685	4

Tabela 3. Resultados do arquivo *z3*, fila igual ao número de threads

Qtd. threads	Tamanho Fila	Tempo por tarefa (média) (ms)	Desvio do tempo por tarefa (ms)	Média por trabalhador	Desvio da média por trabalhador	Qtd. vezes de fila vazia
1	1	49,768000	115,204784	16	0	16
2	2	135,425813	316,311044	8	1	9
3	3	161,097375	342,628050	5,333333	4,189935	6
4	4	188,874250	380,483383	4	2,121320	6
6	6	201,189750	345,966232	2,666667	0,942809	5

Tabela 4. Resultados do arquivo *z3*, fila igual a 4 vezes o número de threads

Qtd. threads	Tamanho Fila	Tempo por tarefa (média) (ms)	Desvio do tempo por tarefa (ms)	Média por trabalhador	Desvio da média por trabalhador	Qtd. vezes de fila vazia
1	4	50,108438	116,273057	16	0	4
2	8	149,474562	345,966331	8	0	3
3	12	179,879312	391,315931	5,333333	4,189935	2
4	16	186,781250	375,387848	4	2,121320	3
6	24	190,438688	314,240887	2,666667	1,247219	3

Numa primeira vista, pode parecer que usar mais threads não compensou, quando se observa apenas o tempo médio por tarefa. Mas há de se levar em conta também o desvio-padrão, o qual também aumentou muito em testes com mais threads. Há de se citar que neste tempo apresentado não há inclusão de qualquer overhead advindo da sincronização, já que esses tempos são apenas para executar a função *fractal*.

Uma análise que pode ser feita é que, com uma fila menor, é esperado que ela seja encontrada mais vezes vazia, o que de fato acontece tanto entre as duas iterações do teste, com uma fila menor e uma maior, quanto dentro de uma mesma iteração, visto que a fila é proporcional à quantidade de threads e, com mais threads, temos uma fila maior, mas uma mesma quantidade de tarefas no arquivo, o que diminui a quantidade de vezes que a fila fica vazia.

Em termos de balanceamento de carga entre as threads, vê-se que ela é similar para os dois arquivos, com uma média próxima do fator $\frac{\text{Quantidade total de Tarefas}}{\text{Quantidade de Threads}}$. O desbalanceamento do arquivo *z3* pode ser notado com os desvios-padrão da carga serem levemente maiores (se comparados às respectivas médias) do que nos testes do arquivo *t*.

Por exemplo, na tabela 1, com 3 threads, cada uma ficou com uma média de cerca de 21, com desvio de 10. Por outro lado, na tabela 3, com 3 threads, cada uma ficou com uma média de aproximadamente 5, mas com desvio-padrão de pouco mais de 4. Comparativamente o desvio-padrão dos testes do *z3* é muito próximo da média, o que não ocorre com os do arquivo *t*.

References

Maziero, C. A. (2019). Problemas clássicos. In *Sistemas Operacionais: Conceitos e Mecanismos*, chapter 12, pages 134–147. DINF - UFPR.

Wikipedia (2021). Mandelbrot set. [1]. Acesso em: 11 dez 2021.

[1] `https://en.wikipedia.org/wiki/Mandelbrot_set`