Trabalho de implementação

Computação Concorrente (MAB-117) — 2021/2

Fuel Assistant - Ferramenta para Cálculo de Gasto de Combustível

Miguel Santos Uchôa da Fonseca

[1]*DRE 120036412*

1. Descrição do problema

O problema escolhido para resolução através da computação concorrente foi uma aplicação do problema de integração trapezoidal e de multiplicação escalar de uma matriz. A ferramenta proposta neste relatório é uma calculadora de gasto de combustível baseada na variação de velocidade por tempo em uma dada estrada e os rendimentos de diferentes carros dispostos em um vetor.

O programa é capaz de observar o comportamento de um carro em uma via através da análise da relação entre velocidade e tempo e descobrir seu deslocamento total através de um método numérico de integração. A partir do conhecimento da extensão da rua através do deslocamento total, o programa então consegue calcular o gasto de combustível de diferentes modelos de carro baseado em seus rendimentos (litros de combustível por quilômetro).

Entradas:

- um arquivo .txt que descreve uma função de velocidade por tempo (em m/s) através da listagem de polinômios e intervalos de tempo (intervalos do eixo x) aos quais se aplicam. A primeira linha do arquivo é a quantidade de polinômios diferentes que descrevem a função. As linhas seguintes são compostas pelo grau de um polinômio, seguido de cada um dos seus coeficientes em ordem decrescente de grau, e por fim o x no qual o intervalo desse polinômio acaba. O programa assume que o primeiro polinômio começa em 0, e que os polinômios seguintes começam no fim de seus anteriores.
- (Opcional) um arquivo .txt que armazena um vetor de rendimentos de carros em km/L de combustível. Cada valor separado por whitespace é o rendimento de um modelo de carro (sendo a primeira linha o rendimento do carro 1, a segunda o rendimento do carro 2 e assim por diante). O programa também cria um vetor de rendimentos aleatórios (para ajudar em testes de desempenho).

Saídas:

- um arquivo .txt que armazena o vetor de combustível gasto (em litros) de cada carro, sendo cada linha o gasto de um carro.

Através da computação concorrente, podemos melhorar bastante o tempo de execução da tarefa descrita, dado que o problema da integração trapezoidal consiste na soma de diversas pequenas áreas de trapézios debaixo de um gráfico, o que poderia ser dividido em somas parciais a serem feitas por cada thread. Além disso, a multiplicação escalar de um vetor também é uma tarefa que pode se tornar custosa, cuja divisão por múltiplas threads também pode ser útil.

2. Projeto e implementação da solução concorrente

Para a implementação da ferramenta, é necessário primeiro observar cada uma das tarefas a serem executadas pela aplicação. Primeiramente, precisamos fazer a leitura da função através de seu arquivo gerador (do qual escolhi perguntar o nome para o usuário assim que executa o programa). Também é necessário ter certeza de que a função é diferenciável, então conferimos isso. Em seguida, geramos o vetor de rendimentos de combustível de forma similar.

Agora, chegamos na primeira parte que utiliza concorrência: calculamos a área debaixo da função dada. Para isso, dividimos essa área em pequenos trapézios de largura delta (valor o qual está definido como ”resolução”no programa, e é inversamente proporcional tanto à precisão do resultado quanto ao tempo de execução). Para fazer a divisão da função em trapézios, havia a opção de tratar a função inteira como uma só, ignorando momentos em que o intervalo de um polinômio acaba e começa outro, ou havia a abordagem de tratar cada intervalo de polinômio como uma função menor, descobrir sua área e depois somar todas. Escolhi a segunda abordagem, visto que seria problemático e custoso tentar descobrir, durante a iteração, em qual intervalo de polinômio um determinado x se encontrava. Quanto a divisão de tarefas, escolhi calcular a área parcial da função inteira de forma intercalada (isto é, cada thread calculando ”trapézio sim trapézio não”até o fim), para depois somar as áreas parciais e obter a integral total. Essa abordagem me pareceu a mais prática, de forma a evitar a necessidade de muita comunicação entre as threads).

Antes de partirmos para a segunda parte concorrente (a multiplicação do vetor), é necessário que a integral da função esteja completamente calculada, e que todas as threads tenham conhecimento dela. Para que nenhuma thread avance antes de isso ser feito, esperamos em uma barreira para que todas terminem seu cálculo. *Obs.:* O resultado da integral total (a soma de todas as somas parciais de cada thread) é armazenado em uma variável global, mas não há necessidade de mutex dado que o resultado deve ser o mesmo para todas (não há risco de inconsistência).

A divisão (ou multiplicação) do vetor de rendimentos a seguir é bem simples de se fazer. Novamente, iteramos pelo vetor de forma intercalada e dividimos o resultado da integral por cada elemento do vetor (lembrando de fazer a conversão de quilômetros para metros). A barreira final é feita pela função `pthread_join`. Por fim, o programa volta para a sua seção consecutiva, mostrando seu tempo de execução, o resultado da integral e criando o arquivo com os gastos de combustível.

Nota-se que cada thread ficará, durante o processo todo, responsável por volta de *1/nthreads* do trabalho, graças a essa divisão intercalada. Todas as abordagens escolhidas (incluindo variáveis globais versus variáveis locais) foram feitas de modo a minimizar - e nesse caso, completamente erradicar - a necessidade do uso de seções mutuamente exclusivas, para diminuir o tempo de execução ao máximo.

3. Casos de teste

Para avaliar a corretude do algoritmo, utilizei a ferramenta externa do *Wolfram Alpha* afim de conferir a precisão das integrações. O resultado dessas integrações ”manuais”se encontra em um comentário no final de cada um dos arquivos de funções de teste. Além disso, um vetor de rendimentos curto, cujo nome é *fuel1.txt,* foi utilizado para testar se as divisões no final estavam sendo feitas corretamente.

Para avaliar o desempenho do algoritmo, existem 3 arquivos de função (denominados *roadX.txt*) em ordem crescente de extensão) e 3 arquivos de vetores de rendimento (denominados *fuelX.txt*) em ordem crescente de extensão).

A primeira função tem apenas 2 polinômios e acaba em x = 6,317. A segunda tem 3, e acaba em x = 906. A terceira tem 4 polinômios e acaba em x = 950.

O primeiro vetor tem 16 elementos, o segundo tem 200 e o terceiro tem 10000.

A resolução escolhida (delta) foi de 0,000001.

Foram realizados um total de 9 testes diferentes - cada função com cada vetor.

4. Avaliação de desempenho

A máquina utilizada foi um MacBook com processador 2,8 GHz Dual-Core Intel Core i5 de 2014, com 4 núcleos de execução, rodando MacOS Catalina 10.15.7.

Cada teste foi realizado pelo menos 5 vezes, das quais o menor tempo foi retirado. Seguem os resultados obtidos:

Testes	1 thread	2 threads	(aceleração)	4 threads	(aceleração)
r1-f1	1,053s	0,536s	1,964	0,387s	2,720
r1-f2	1,053s	0,540s	1,950	0,400s	2,632
r1-f3	1,055s	0,538s	1,960	0,392s	2,691
r2-f1	23,450s	12,236s	1,916	11,143s	2,104
r2-f2	23,604s	12,756s	1,850	11,409s	2,068
r2-f3	25,411s	12,455s	2,040	11,448s	2,219
r3-f1	36,117s	18,594s	1,942	15,453s	2,337
r3-f2	36,119s	18,710s	1,930	15,418s	2,342
r3-f3	36,089s	18,671s	1,932	15,405s	2,343

5. Discussão

Observa-se que o tamanho dos vetores não tem muito efeito sobre o tempo de execução (provavelmente pelo seu tamanho menor), enquanto a complexidade e o tamanho das funções representam grande parte do tempo de cálculo.

Para 2 threads, a aceleração obtida foi a esperada: por volta de 2. Para 4 threads, no entanto, o tempo de execução foi bem longe do esperado (por volta de 4 vezes mais rápido que com 1 thread). Isso certamente se dá pelo fato do processador da máquina utilizada ter exatamente 4 threads, o que significa que o programa tem que competir com outros processos do computador e dificilmente utilizará as 4 threads ao mesmo tempo. Mesmo assim, o tempo de execução melhorou levemente em todos os casos.

Para melhorar o programa, acredito que a experiência de usuário poderia melhorar - uma maneira de armazenar configurações para não ter que ficar sempre selecionando os arquivos de input e output seria bem útil. Além disso, uma maneira de configurar a resolução dentro do programa seria útil e fácil até de implementar (só não foi implementado pra não haver outra pergunta a ser respondida durante a execução). Por último, pensei também em implementar uma ordenação dos carros por uso de gasolina (do menor pro maior), mas foquei nas duas partes mais cruciais da aplicação - principalmente pois isso

seria equivalente a ordenar os rendimentos logo de cara, e seria independente de estrada (função).

Para melhorar os testes, gostaria de ter usado uma máquina com processador melhor, mas não tenho nenhuma disponível no momento. Além disso, seria interessante ver números maiores de elementos nos vetores, mas como não existem tantos modelos de carro no mundo e o meu foco principal com o programa era a integração e a implementação de barreira, preferi limitar a 10 mil carros mesmo.

Desde o início estava interessado em fazer uma implementação envolvendo integração, mas demorei para conseguir pensar em uma boa ”aplicação no mundo real”(ou no mínimo algo pra fugir da abstração). Além do mais, não queria ficar preso a uma aplicação que utilizasse apenas soma intercalada e possuísse outros métodos de sincronização - por isso a divisão no vetor final.

6. Referências bibliográficas

https://en.wikipedia.org/wiki/Trapezoidal_rule, acesso em 31/01/2022.

https://www.desmos.com/calculator, acesso em 31/01/2022.

https://www.wolframalpha.com, acesso em 31/01/2022.