Universidade Federal do Rio de Janeiro

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

MAYARA MARTINS POIM FERNANDES

TOMAZ CUBER GUIMARÃES

RELATÓRIO: IMPLEMENTAÇÃO CONCORRENTE DO MÉTODO DE INTEGRAÇÃO NUMÉRICA RETANGULAR

Rio de Janeiro

2019

SUMÁRIO

1. INTRODUÇÃO

Neste trabalho analisamos a implementação em linguagem C do método de integração numérica retangular (com ponto médio) em duas versões: uma sequencial e outra concorrente. A versão concorrente usa a biblioteca POSIX Threads para implementação dos fluxos de execução (*threads)*. Nossa análise consiste em medir o tempo de execução de cada versão para um conjunto definido de funções de teste, variando dentre esses o intervalo de integração e o erro máximo permitido. Para a versão concorrente, também medimos o tempo de execução para número diferentes de threads, visando calcular o ganho de desempenho em cada caso.

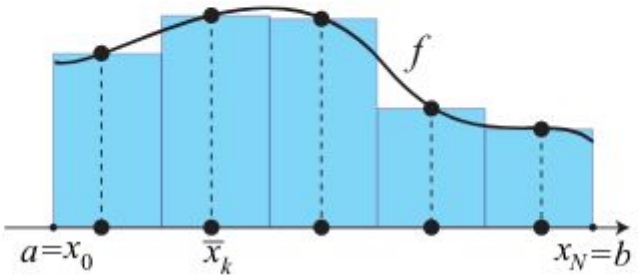
2. O PROBLEMA

O método de integração numérica é uma ferramenta necessária para quando precisamos calcular a integral de uma função $\int_a^b f(x)\,dx$ e não conhecemos uma antiderivada (nesse caso, não é possível aplicar o Teorema Fundamental do Cálculo) ou não sabemos a expressão de f (como no caso de dados coletados em laboratório projetados em um gráfico). A saída que ele nos possibilita é calcular uma aproximação numérica para a integral.

O método escolhido foi o de aproximação retangular com ponto médio. Esse método consiste em dividir o intervalo de integração escolhido $[a, b]$ em vários sub-intervalos, integrar $f(x)$ em cada sub-intervalo e somar todos os valores para obter o valor da integral nesse intervalo. A divisão do intervalo é realizada a partir de **m**, ponto médio do segmento equivalente ao intervalo inicial. Em seguida,

Dados $f(x) \in C([a, b])$ e N sub-intervalos em $[a, b]$ de comprimento $h = \frac{(b-a)}{N}$ com $x_0 = a$ e $x_n = b$, temos:

$$\int_{x_0}^{x_N} f(x)\,dx \approx h \sum_{k=1}^{N} f(\bar{x}_k) \quad \text{com} \quad \bar{x}_k = \frac{x_{k-1} + x_k}{2}$$



3. ABORDAGENS

Nesta seção, descreveremos como escolhemos abordar a implementação do método tanto no caso **sequencial** (seção 3.1) quanto no caso **concorrente** (seção 3.2). Em ambos os casos, foi utilizado o arquivo de *header* **funcoes.h** que contém as declarações das funções passadas no enunciado (implementadas em **funcoes.c)** que foram usadas como casos de teste.

(b) $f(x) = \sqrt{1 - x^2}, -1 < x < 1$
(c) $f(x) = \sqrt{1 + x^4}$
(d) $f(x) = sen(x^2)$
(e) $f(x) = cos(e^{-x})$
(f) $f(x) = cos(e^{-x}) * x$
(g) $f(x) = cos(e^{-x}) * (0.005 * x^3 + 1)$

Legenda: lista de funções incluídas nos casos de teste.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

long double f1(long double x){
        return 1 + x;
}

long double f2(long double x) {
        if ((x < 1) && (x > -1))
        return sqrt(1 - pow(x,2));
        else{
        printf("Argumento inválido para f2: -1 < x < 1\n");
        exit(-1);
        }
}

long double f3(long double x){
```

```
        return sqrt(1 + pow(x,4));
}

long double f4(long double x){
        return sin(pow(x,2));
}

long double f5(long double x){
        return cos(pow(M_E,-x));
}

long double f6(long double x){
        return cos(pow(M_E, -x)) * x;
}

long double f7(long double x){
        return cos(pow(M_E, -x)) * (0.005 * (pow(x,3) + 1));
}
```

Legenda: Implementação das funções no arquivo funcoes.c.

Para executar os programas, o usuário deve passar como argumento pela linha comando o **início e o fim do intervalo** de integração [a,b] , o **erro máximo** permitido *e* e a **função** a ser integrada. Na versão **concorrente**, o **número de threads**, também deve ser informado.

É importante ressaltar que utilizou-se **long double** para lidar com os intervalos e erros com o objetivo de garantir uma precisão teoricamente máxima.

Para correlacionarmos a função de entrada com as implementações do arquivo **funcoes.c** foi criada uma variável global **long double** (*funcao)(**long double**), um ponteiro para função, que, após o usuário informar a função escolhida, recebe o endereço dela que está armazenado no vetor **long double** (*funcoes[7])(**long double**). Um vetor de endereços de funções, e um ponteiro global de função foram utilizados visando simplificar o processo de seleção e passagem da função escolhida.

```
long double (*funcoes[7])(long double) = {&f1, &f2, &f3, &f4, &f5, &f6, &f7}; //
vetor com todas as possibilidades de funcoes
// forcar usuario a escolher um intervalo, erro e a funcao
if(argc < 5) {
       printf("<inicio do intervalo> <fim do intervalo> <erro permitido> <funcao
a ser integrada>\n");
        printf("(f1) f(x) = 1 + x\n(f2) f(x) = √(1 - x^2), -1 < x < 1\n(f3) f(x)
= √(1 + x^4)\n(f4) f(x) = sen(x^2)\n(f5) f(x) = cos(e^(-x))\n(f6 ) f(x) =
cos(e^(-x)) * x\n(f7) f(x) = cos(e^(-x)) * ((0.005 * x^3) + 1)\n");
       exit(-1);
}
inicio = strtold(argv[1], NULL);
fim = strtold(argv[2], NULL);
erroPermitido = strtold(argv[3], NULL);
escolha = argv[4];

// condicional para forcar o usuario a escolher uma funcao valida
if(escolha[0] != 'f' || escolha[1] > '7' || escolha[1] < '1') {
```

```
        printf("Insira uma funcao valida\n");
        exit(-1);
}

 // operacao para atribuir a funcao global a funcao escolhida
funcao = funcoes[(escolha[1] - '0') - 1];
```

Legenda: validação da entrada do usuário no caso sequencial.

3.1 SEQUENCIAL

A implementação sequencial, após a validação da entrada do usuário, realiza uma chamada a função **long double integracaoRetangular**(**long double** inicio, **long double** fim). Essa função calcula recursivamente a integral numérica da função escolhida pelo método da aproximação retangular usando a Regra do Ponto Médio. Foi escolhida uma abordagem de recursão devido ao perfil altamente recursivo do problema, observando-se que tentar uma abordagem distinta poderia trazer uma complicação desnecessária à solução sequencial.

Para isso utilizar-se dessa Regra, definimos uma função auxiliar **long double pegarPontoMedio**(**long double** inicio, **long double** fim), que dá o ponto médio de um intervalo passado como argumento.

A função principal calcula primeiramente, o ponto médio **m** do intervalo de integração **[a,b]** e o valor ***f*(m)** da função que será integrada nesse ponto. Em seguida, calcula os pontos médios **e** e **d** dos segmentos **[a,m]** e **[b,m]**, respectivamente, e os valores de ***f*(e)** e ***f*(d)**.

Com isso, temos os valores para as áreas dos retângulos principal ($(b-a)\cdot f(m)$), da esquerda ($(m-a)\cdot f(e)$) e da direita ($(b-m)\cdot f(d)$) e podemos calcular o erro como sendo o módulo da diferença entre a área do retângulo principal e a soma das áreas dos retângulos menores. Caso esse erro seja maior que o erro permitido, a função realiza duas

chamadas recursivas a si mesma, uma no intervalo **[a,m]** e outra no intervalo **[b,m]**, sendo o valor da integral igual a soma dos retorno de ambas as chamadas. Caso contrário, a função retorna o valor da área do retângulo principal como valor da integral no intervalo.

Essa implementação garante a **quadratura adaptativa** baseada no valor do erro máximo dado: a quantidade **N** de subintervalos em que se dividirá o intervalo de integração é definida de acordo com o número de chamadas recursivas à função **integracaoRetangular**.

3.2 CONCORRENTE

No processo de debate e reflexões sobre como paralelizar este processo recursivo esbarrou-se em diversos problemas, dentre eles: Como lidar com esse perfil de recursividade de gerar novas chamadas a função de integrar? Como conseguir balancear as tarefas entre as threads de forma satisfatória?

Para lidar com a primeira questão, pensou-se em fazer com que uma thread faça o processamento de um intervalo e realize a comparação com o erro, da mesma forma que o sequencial. Caso satisfaça o erro permitido, ela insere esse resultado em um espaço dedicado a ela em um vetor de resultados. Caso contrário (ou seja, ela entre na situação em que haviam as chamadas recursivas), essa thread vai precisar agora distribuir esses novos intervalos, que seriam as tarefas a serem realizadas/processadas pelas threads.

Percebeu-se que uma abordagem poderia ser um uso de algum tipo de **buffer** para armazenar esses intervalos a serem processados (ou seja, as tarefas). Para facilitar organização e a passagem desses intervalos, criou-se uma struct:

```
typedef struct _Intervalo {
       long double inicio;
       long double fim;
} Intervalo;
```

Essa ideia de ter um buffer, e ter threads retirando e colocando tarefas nele se assemelha a um problema Produtor-Consumidor. A partir daqui, trataremos o problema como tal, havendo o diferencial de todas as threads serem produtoras **e** consumidoras.

Começamos criando a função das threads `void *calculaIntegral(void *args).` A partir daí, foi necessário refletir sobre qual seria o critério de parada das threads. Concluiu-se que o processamento de cada thread pode parar quando: (a) não temos mais intervalos a serem

processados no buffer (buffer vazio); e (b) não temos mais threads trabalhando (ou seja, não existe a possibilidade de um intervalo ser inserido no buffer).

Para checar (a), usamos de uma variável global `proximo` que mantém o controle de qual a próxima posição para inserção. Para (b), criamos uma variável global `threadsTrabalhando` que conta quantas threads estão trabalhando. Essas checagens são então feitas na função das threads. Uma vez feito isso, a thread remove um intervalo do buffer, checa sua validade, e depois chama a função de integração retangular para realizar o processamento do intervalo.

```
while(1) {
        pthread_mutex_lock(&mutex);
        if(proximo == 0 && threadsTrabalhando == 0) {
             // a condicao de parada das threads consiste em nao ter mais intervalos pendentes na pilha
             // E nao ter mais threads trabalhando (ou seja, nao tem a possibilidade de algo ser colocado na pilha)
              pthread_cond_broadcast(&cond_rem); // liberar threads presas no wait de remocao (do contrario, elas ficariam presas ali)
             pthread_mutex_unlock(&mutex);
             break;
         }
         pthread_mutex_unlock(&mutex);
         intervalo = removeBuffer(); // pega um intervalo da pilha
         if(intervalo.inicio == 0 && intervalo.fim == 0) { // se for invalido, sai do loop
             break;
         }
         integracaoRetangular(intervalo, id); // processa esse intervalo
         pthread_mutex_lock(&mutex);
         threadsTrabalhando--;
         pthread_mutex_unlock(&mutex);
}
```

Observa-se que foi necessário fazer adaptações no padrão Produtor-Consumidor para se adequar às particularidades do problema. Por exemplo, uma thread pode entrar na checagem usual para chamada de um wait na função de remover do buffer, mas não existir a possibilidade de inserção de novos intervalos no buffer. Nesse caso, optou-se pela função retornar um intervalo inválido, e isso ser tratado na função das threads.

A função de processamento dos intervalos possui uma lógica similar à da implementação sequencial. Mas agora, ao invés de chamadas recursivas, as threads irão distribuir as tarefas, como mencionado anteriormente. E ao invés de um retorno direto com a integral nos casos em que o erro é menor, teremos uma adição na posição dedicada à thread no vetor de resultados.

É importante observar que o uso de um buffer para esse controle aumenta bastante a contenção do problema. Isso é uma consequência consciente da abordagem escolhida. Porém, é interessante tentar minimizar um pouco esse custo. Para tanto, optou-se que as threads, ao invés de inserirem os dois novos intervalos no buffer, peguem o intervalo da esquerda e façam elas mesmas o processamento (uma nova chamada da função), e coloquem somente o intervalo da direita no buffer, reduzindo as idas ao buffer.

```
   if(erro > erroPermitido) { // caso erro seja maior do que o permitido
        Intervalo temp;
        temp.inicio = inicio;
        temp.fim = meio;
        integracaoRetangular(temp, id); // thread pega o intervalo da "esquerda" e
ela mesma processa
        temp.inicio = meio;
        temp.fim = fim;
        insereBuffer(temp); // coloca o intervalo da "direita" na pilha
    }
    else resultados[id] += areaTotal; // caso seja um erro aceitavel, soma essa
area aos seus resultados
```

Legenda: seção com a lógica do processamento pós-checagem do erro

Por ser um problema de produtor-consumidor, também temos implementações das funções de inserção e remoção.

Optou-se por buffers grandes, para evitar que haja retenção na inserção do buffer (threads precisando esperar abrir um espaço para conseguirem inserir).

Inicialmente, esse buffer foi pensado como uma pilha, mas isso acarretava em um problema de inconsistência na saída do programa, mesmo se fosse rodado com as mesmas entradas. Isso ocorre porque não se garante uma consistência na ordem em que cada thread tirava ou inseria um intervalo da pilha. Dessa forma, a soma final ocorria em ordens

inconsistentes. E por se tratar de um somatório de ponto flutuante, isso afetava a consistência dos resultados.

Para lidar com esse problema, pensou-se em substituir a pilha única por filas (com objetivo de se aproximar melhor do problema usual de produtor-consumidor) individuais para cada thread. Agora, cada thread remove um intervalo de sua fila, e insere na fila da thread "seguinte" (id seguinte, sendo de forma circular). Por exemplo, em uma situação com 4 threads, a thread 3 insere na fila da 0. Dessa forma, conseguimos garantir uma consistência nessa ordem de processamento, e consequentemente na ordem das somas.

```
tid = (id + 1) % nthreads;
insereBuffer(temp, tid); // coloca o intervalo da "direita" na fila da thread
seguinte
```

Voltamos agora ao segundo questionamento levantado no início desta seção. Como essas implementações garantem um balanceamento?

Em uma abordagem similar aos métodos para identificação de primos abordados em sala, nós sempre teremos threads pegando tarefas, enquanto houver tarefas sendo geradas. No entanto, é importante observar que devido a escolha desse método, existe uma dependência das threads estarem *gerando* trabalho. Em algumas situações, isso poderia gerar um desbalanceamento, mas nos casos abordados, a solução proposta garantiu um equilíbrio julgado apropriado.

Na solução das pilhas, por ter uma lógica mais diretamente similar à situação dos primos, esse balanceamento ficou aparentemente mais claro. Mas na abordagem de filas, restaram algumas dúvidas sobre um possível efeito de uma thread depender especificamente da thread de id anterior gerar tarefas. Para averiguar isso, avaliou-se o balanceamento com um contador de "serviço" para cada thread, e nos casos testados, os resultados foram bem similares aos da solução de pilha, mostrando que utilizar uma fila para cada thread **também** oferta um balanceamento apropriado, além de garantia na consistência na soma.

4. ANÁLISE DE DESEMPENHO

4.1 METODOLOGIA

Definimos os casos de teste de cada versão do programa (sequencial, concorrente com pilha e concorrente com fila) como variações no tamanho do intervalo, magnitude do erro, função escolhida e, nos casos concorrentes, o número de threads. Essas variações foram:

	Pequeno	**Médio**	**Grande**
Intervalo	[-0.9 , 0.9]	[-10 , 10]	[-100 , 100]

	Alto	**Médio**	**Baixo**
Erro máximo	10^{-4}	10^{-9}	10^{-19}

Número de Threads	1	2	4	8

Um dos problemas reparados ao se pensar em casos de teste para os programas implementados foi o grande volume de execuções necessárias para testar caso a caso todos as combinações possíveis de intervalos, funções, erro máximo permitido e, para as

implementações concorrentes, número de threads. A solução escolhida foi **automatizar** a coleta de resultados para os casos de testes escolhidos. Isso foi realizado utilizando a linguagem de *scripting* **Bash Script** para rodar iterativamente os programas variando os argumentos a cada execução. Esses *scripts* rodam todos os casos definidos para cada versão e salvam a saída em um arquivo .csv que depois foi importado para uma planilha contendo todos os resultados das execuções.

```
#!/bin/bash
#Script para automatizar testes da versão sequencial do programa de integração
numérica

array_min_intervalo=(-0.9 -10 -100)
array_max_intervalo=(0.9 10 100)
array_erro=(0.0001 0.000000001 0.0000000000000001)
#erro=0.0001

echo "função,intervalo,erro,integral,tempo" > teste.csv

for i in $(seq 1 ${#array_min_intervalo[@]}); do

	min_intervalo=${array_min_intervalo[i-1]}
	max_intervalo=${array_max_intervalo[i-1]}
	echo "Intervalo atual: [$min_intervalo, $max_intervalo]"

	for erro in "${array_erro[@]}"; do
		echo "Erro atual: $erro"

		for j in $(seq 1 6); do
			echo "Função atual: f$j"
			output=$(./sequencial $min_intervalo $max_intervalo $erro
f$j)
			echo "f$j,[$min_intervalo, $max_intervalo],$erro,$output"
>> teste.csv
		done
	done
done

#!/bin/bash
#Script para automatizar testes da versão concorrente com fila do programa de
integração numérica

array_min_intervalo=(-0.9 -10 -100)
array_max_intervalo=(0.9 10 100)
array_erro=(0.0001 0.000000001 0.0000000000000001)
#erro=0.0001
```

```
echo "função,número de threads,intervalo,erro,integral,tempo" > teste_fila.csv

for i in $(seq 1 ${#array_min_intervalo[@]}); do

       min_intervalo=${array_min_intervalo[i-1]}
       max_intervalo=${array_max_intervalo[i-1]}
       echo "Intervalo atual: [$min_intervalo, $max_intervalo]"

       for erro in "${array_erro[@]}"; do
           echo "Erro atual: $erro"

           for (( nthreads=1; nthreads<=8; nthreads*=2)) do
                 echo "Número de Threads atual: $nthreads"
                 for j in $(seq 1 6); do
                       echo "Função atual: f$j"
                       output=$(./concorrenteFila $min_intervalo $max_intervalo
$erro $nthreads f$j)
                       echo "f$j,$nthreads,[$min_intervalo,
$max_intervalo],$erro,$output" >> teste_fila.csv
                 done
            done
      done
done
```

4.2 SEQUENCIAL

Foi observado que a partir do caso

4.3 CONCORRENTE

5. CONCLUSÕES

Este trabalho se mostrou um excelente desafio para exercitarmos os conhecimentos dos padrões estudados, assim como de uso de locks e condições. Lidar com uma situação altamente recursiva, e com a necessidade de se estabelecer um balanceamento trouxe mais dificuldades do que inicialmente pensado. A medida que foram estabelecidas soluções para um problema levantado no âmbito concorrente, encontrava-se novos impedimentos. No final, chegamos em 2 soluções satisfatórias, mas com seus problemas individuais, decorrentes de escolhas de implementação explicadas ao longo do relatório.

6. REFERÊNCIAS BIBLIOGRÁFICAS

1. S. C. COUTINHO, *Cálculo Numérico,* Departamento de Ciência da Computação, Instituto de Matemática, Universidade Federal do Rio de Janeiro, 2018

2. A. PAIVA *Integração Numérica,* Departamento de Matemática Aplicada e Estatística, Instituto de Ciências Matemáticas e de Computação, Universidade do Estado de São Paulo - São Carlos, 2014

3. Repositório contendo o código-fonte do trabalho. Disponível em: <https://github.com/mayara21/Comp-Conc-Integracao-Numerica>.

4. Planilha com Resultados das Execuções dos casos de teste. Disponível em: <https://docs.google.com/spreadsheets/d/1ihakpWuf0Bgvjb--3IFLJBb7v5YWsUzml3b4zqnmNtg/edit?usp=sharing>.