# Trabalho de implementação

Computação Concorrente (MAB-117) — 2021/2

# Integração numérica retangular utilizando programação concorrente

Leonardo Ribeiro Santiago (120036072) & Caio Gomes Monteiro (120036373)

## 1 Descrição

O programa implementado calcula a integral numérica de um polinômio de grau e coeficientes abritrários, em um intervalo.

Para executar, o usuário deve passar na linha de comando:

1. o começo do intervalo de integração
2. o fim do intervalo de integração
3. o número de divisões do intervalo (quantos retângulos terá no meio)
4. o número de threads

Após, também é necessário digitar no *stdin* o grau do polinômio, e os coeficientes em ordem (primeiro o de grau 0, depois grau 1, etc).

Essencialmente, o programa aproxima a área embaixo da função como diversos retângulos de base $\Delta x$ e altura $f(x)$. Assim, como o cálculo de cada área é independente das outras, podemos quebrar o problema em várias threads, somar um grupo de retângulos em cada uma das threads e por fim somar todas as áreas encontradas.

## 2 Projeto e implementação

Precisaremos calcular a área de $n$ retângulos. Assim, poderíamos dividir $n$ pelo número de threads e passar cada sequência contígua de retângulos para a threads. Entretanto, se o número de threads não for um divisor exato de $n$, precisaríamos de uma heurística para assinalar o resto dos retângulos à alguma thread (por exemplo, o resto é calculado pela última thread). Isso faria com que uma das threads fizesse mais cálculo do que as outras.

Ao invés disso, escolhemos fazer com que cada thread calcule apenas os retângulos de índice igual ao seu `id` módulo número de threads. Essa decisão faz com que a quantidade de trabalho seja mais uniformemente distribuida entre cada thread.

Para tal, é necessário calcular a base dos retângulos $\Delta x$, utilizando as informações dadas pelo usuário. Depois, basta calcular a área utilizando um laço `for`. Para garantir que os retângulos sejam distribuidos corretamente, iniciamos eles no índice igual ao `identificador` da thread, e depois somamos o número de threads.

# 3 Corretude e testes

O programa implementado apenas aproxima a área da integral definida como a soma da área de retângulos embaixo da curva. Assim, é esperado que, conforme o número de divisões aumente, o erro do programa diminua.

Para calcular o erro, utilizamos o Sympy (Meurer et al. 2017) para calcular o valor da integral definida no intervalo (de maneira exata), e depois simplesmente achamos a diferença absoluta entre o valor exato e o valor da aproximação.

Teoricamente, o valor achado deveria ser igual independentemente da quantidade de threads. Entretanto, como estamos trabalhando com floats, sabemos que a soma e a subtração não são associativas, então a ordem que essas operações são feitas podem de fato alterar o resultado obtido.

Assim, também precisamos testar se o erro achado para cada número de threads está parecido. Para tal, grafamos o erro para o número de divisões e para cada número de threads (Figure 1); como esperamos que todas estejam próximas, trocamos o marcador baseado na quantidade de threads e vemos se a tendência geral conforme aumentamos o número de retângulos é que o erro diminua em todas as threads.

Assim, escolhemos alguns polinômios de teste, calculamos o valor exato e mostramos o erro conforme aumentamos o número de retângulos. É possível notar que nos casos em que temos área negativa (e portanto soma de números negativos), a não-associatividade dos números *floating-point* fica bem mais clara, e podemos ver uma diferença nos resultados variando o número de threads. De fato, essa diferença está presente em todos os elementos, mas fica muito mais gritante quando temos números negativos.

Ainda sim, vemos que a tendência geral do erro de todos os polinômios é diminuir logaritmicamente (a tendência natural deste algoritmo de integração), portanto vemos que a implementação do programa está correta, visto que é um programa que não necessariamente busca a resposta exata, mas sim a melhor aproximação para o resultado.

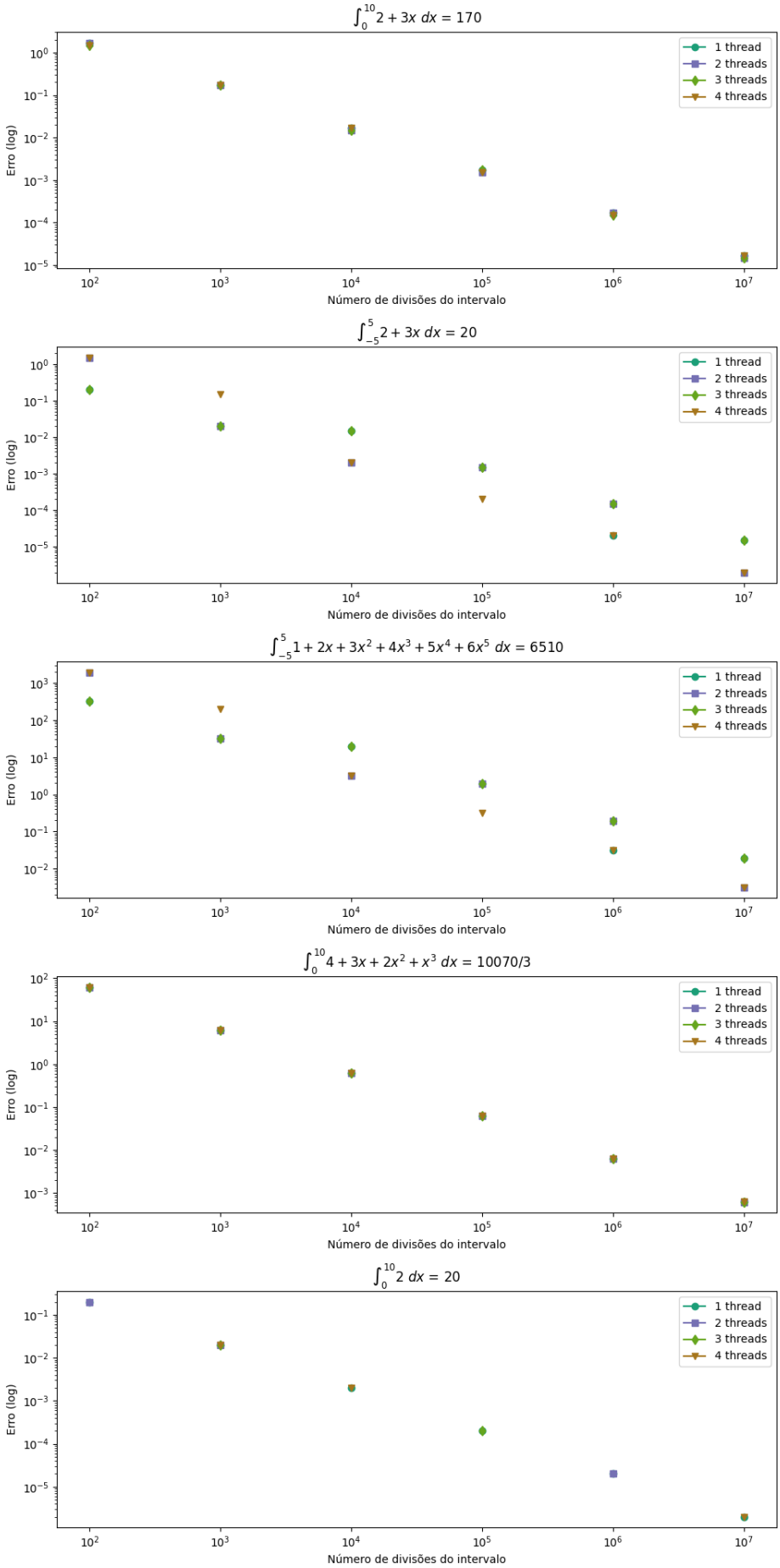


Figure 1: Gráfico de erro por número de divisões no intervalo para cada polinômio de teste.

# 4 Desempenho

Para medir o desempenho do programa, comparamos a quantidade de threads com a quantidade de divisões em cada polinômio. Esperamos que, conforme aumentamos o número de threads e mantemos o número de divisões constante, o tempo de execução caia.

De fato, é isso que vemos nos gráficos na figura 2. Do lado esquerdo, construimos o gráfico de tempo por número de divisões, para cada um dos polinômios e cada um dos números de threads. Utilizamos a escala logarítimica no eixo $y$.

Vemos que conforme o número de divisões aumenta, a reta que representa o programa com 4 threads consistentemente termina abaixo das outras. Vemos também que para números menores de divisões, os programas que rodam com números menores de threads roda muito mais rápido do que os com mais; e conforme o número aumenta, os programas com mais threads passam a dominar o desempenho.

Nos gráficos do lado direito, medimos a quantidade de ganho de desempenho. Como decidimos não implementar o programa sequencial (sem threads), comparamos os tempos atingidos com o programa que roda somente com 1 thread.

Novamente, esses gráficos deixam claro que o *overhead* gerado pelas threads faz com que em quantidades baixas, o programa rode mais rápido com menos threads. Ao aumentar as divisões, vemos que o programa com 4 threads chega a até 400% de ganho de desempenho, ou seja, roda 4 vezes mais rápido.

# 5 Discussão

O ganho de desempenho foi exatamente o esperado, visto que esperamos que as 4 threads executem concorrentemente 1/4 do trabalho, e portanto levem por volta de 1/4 do tempo para terminar. Portanto, quando vemos um ganho de desempenho de 400%, sabemos que o programa está de fato funcionando como deveria.

No que tange à concorrência, não vemos como melhorar o programa (para ganhar desempenho). Entretanto, poderíamos adicionar alguma maneira do programa de medir o erro máximo (utilizando as fórmulas vistas em cálculo numérico) para facilitar o uso para o usuário. Julgamos que tal implementação seria um pouco supérflua, e preferimos não fazê-la por agora.

Outro problema na medição do erro é a soma dos floats, visto que a ordem que a soma executada realmente muda o resultado da soma. Esse erro é muito mais difícil de estimar, visto que depende da quantidade de somas feitas e de muitos outros fatores. Resolvemos também tentar não tocar nesse assunto já que não tange diretamente à concorrência, entretanto essa seria uma das possíveis melhorias do nosso programa (medir esse erro mais precisamente).

Por fim, vale notar que as ferramentas de teste de corretude e desempenhos foram implementadas como um script em python, disponível no repositório do github do código fonte, chamado `testes.py`. Esse arquivo foi implementado de maneira a facilitar adicionar novos casos de teste

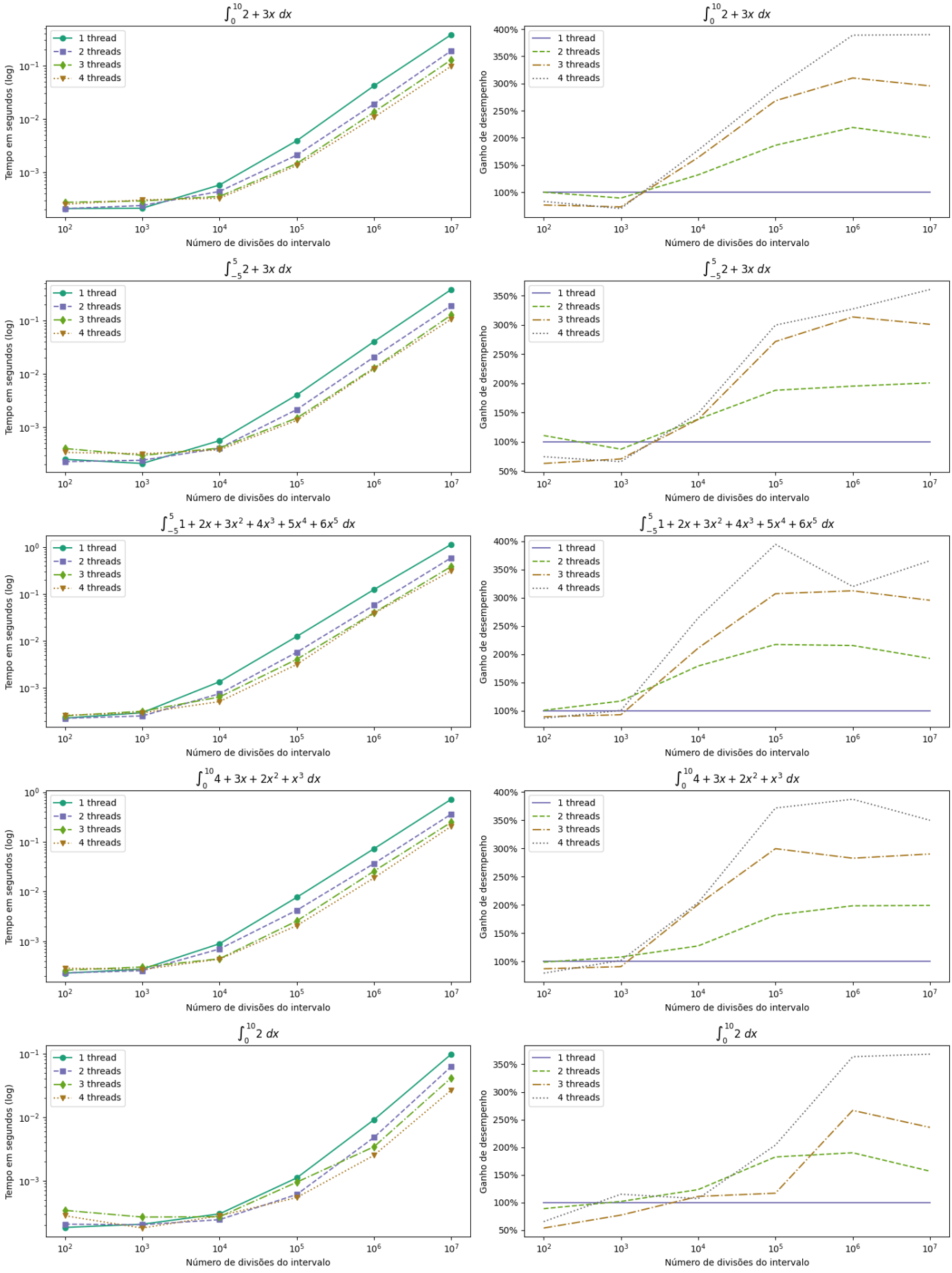


Figure 2: Gráficos de tempo por número de divisões no intervalo (esquerda) e ganho de desempenho por número de divisões (direita).

(novos polinômios, intervalos, etc), bem como modificar como testamos o programa. Também facilita auditar erros nas medições.

# 6 Referências bibliográficas

1. Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. (2017) SymPy: symbolic computing in Python. PeerJ Computer Science 3:e103 `https://doi.org/10.7717/peerj-cs.103`