Ministério da Educação
Universidade Federal do Rio de Janeiro

Instituto de Matemática - IM
Ciência da Computação

Universidade Federal
do Rio de Janeiro

Matheus Abrantes
Paloma Calado

RELATÓRIO DE COMPUTAÇÃO CONCORRENTE -

PROBLEMA DE QUADRATURA PARA INTEGRAÇÃO NUMÉRICA

RIO DE JANEIRO

1. Introdução

A integral definida é uma ferramenta essencial em diversas áreas para definir quantidades como áreas, volumes, pesos, probabilidades, etc.. Entretanto, existem situações nas quais é impossível encontrar o valor exato de uma integral.

Neste relatório apresentaremos o método da quadratura adaptativa que é utilizado para os casos que não sabemos a antiderivada ou até mesmo uma função que modela um comportamento que não é muito estudado e não é possível calcular a área utilizando o Teorema Fundamental do Cálculo.

2. Objetivo

Implementar duas soluções, uma sequencial e outra concorrente, para o problema da quadratura, e analisar o ganho de desempenho obtido.

3. Headers

Para organizar melhor o projeto, criamos dois headers: estruturas.h e functions.h.

- **estruturas.h**

O header estruturas.h, contém a nossa estrutura de dados usada no programa, que é uma pilha implementada de duas maneiras: uma com um array, e outra com uma lista encadeada.

A pilha com a lista encadeada, permite que o programa execute sem ter que se preocupar com o tamanho da pilha, pois o tamanho é dinâmico, ou seja, aumenta e diminui conforme a demanda do programa. Essa abordagem tem a vantagem de economizar memória, pois evita de termos de alocar espaço extra que não vamos usar. Porém, temos o overhead de ter que alocar espaço na memória toda vez que um novo elemento for adicionado na pilha, que consequentemente causa uma perda de desempenho.

Já na versão implementada com array, temos que definir antes, o tamanho do

array, e esse tamanho é estático, porém conforme as threads vão consumindo o conteúdo do array, essas posições são liberadas para um novo valor ser adicionado. Devido a essa natureza temos que atribuir um tamanho que vamos achar suficiente para a execução do programa e torcer pra tudo dar certo. Nessa abordagem quase sempre haverá espaço não utilizado no array. A vantagem dessa abordagem é não precisar chamar a função malloc() para todo elemento que for adicionado no array, então temos um pequeno ganho no desempenho comparado à abordagem com lista encadeada.

Ainda nesse header, implementamos uma estrutura chamada “parametro”, que é formada por: um ponteiro para a função “f”, o limite a esquerda “a”, o limite a direita “b”, e o erro tolerável “erro_max”. Essa estrutura que é passada como parâmetro para nossa função quad_adaptativa() - que é a função que faz o cálculo do intervalo dado com respeito ao erro máximo - e também é o tipo do conteúdo de cada nó da pilha, que é formado por um elemento do tipo “parametro” e um ponteiro para o próximo elemento.

Na implementação da pilha com lista encadeada temos os seguintes métodos:

- ➔ push() - recebe um elemento e o adiciona a pilha.
- ➔ pop() - remove o topo da pilha e retorna o conteúdo.

Na implementação com array temos os métodos:

- ➔ push_arr() - recebe um elemento e o adiciona a pilha.
- ➔ pop_arr() - remove o elemento no topo e o retorna.
- ➔ esta_vazia() - retorna “true” se a pilha está vazia, e “false” caso contrário.
- ➔ esta_cheia() - retorna “true” se a pilha estiver em sua capacidade máxima, e “false” caso cotrário.

- **functions.h**

O header functions.h, foi criado para definir e enumerar as funções cujas integrais queremos calcular, e algumas funções auxiliares na implementação das nossas soluções. São elas:

1. $f(x) = 1 + x$
2. $f(x) = \sqrt{1 - x^2},\ -1 < x < 1$
3. $f(x) = \sqrt{1 + x^4}$

4. $f(x) = sin(x^2)$
5. $f(x) = cos(e^{-x})$
6. $f(x) = cos(e^{-x}) * x$
7. $f(x) = cos(e^{-x}) * (0.005 * x^3 - 1)$

As funções auxiliares são:

- ➔ atribui_parametros() - como o nome indica, recebe como parâmetros os mesmos parâmetros usados para definir a entrada do cálculo de um intervalo - os limites e o erro -, e também um inteiro que será usado para indicar o número da função f(x) desejada, e retorna uma struct com os parâmetros.
- ➔ atribui_parametros_auxiliar() - usada como função auxiliar à anterior, recebe como argumentos: um ponteiro para uma função f(x), limite a esquerda, limite a direita, e o erro, e o atribui a uma variável do tipo struct parametro. Essa função foi criada porque depois que declaramos uma variável com uma struct como tipo, não é possível alterar esses valores de forma sucinta, o que deixaria a função atribui_parametros() muito longa.

4. Soluções implementadas

Conseguimos implementar os dois tipos de solução - sequencial e concorrente - para o problema. A versão sequencial foi bem direta, implementamos a solução recursiva como estava descrita no enunciado do trabalho. Mas tivemos problemas quando fomos implementar a solução concorrente. Implementamos primeiro a solução que usava lista encadeada, e quando fomos comparar o desempenho com a sequencial, a concorrente apresentou perda expressiva no desempenho, incluindo a execução usando somente uma thread. Por esta razão, e também por motivação de que o objetivo deste trabalho, e um dos objetivos da disciplina, era melhorar o desempenho comparado a sequencial, decidimos então implementar mais duas soluções concorrentes: uma bem parecida, usando array, e outra usando um algoritmo diferente, que não garantia balanceamento de carga entre as threads. Falaremos sobre elas mais a frente.

- **Sequencial**

O algoritmo para a solução sequencial funciona da seguinte maneira:

- ➔ Guardamos o input do usuário em uma struct.
- ➔ Chamamos a função quad_adaptativa() com essa struct como parâmetro.
- ➔ A função quad_adaptativa() então calcula o ponto do meio dos limites.
- ➔ Calcula o ponto do meio entre o limite da esquerda, e o ponto do meio do retângulo maior, e depois faz o mesmo com o limite da direita.
- ➔ Calcula o valor da função nesses pontos médios, que serão usados como as alturas dos retângulos.
- ➔ Calcula a área dos retângulos multiplicando a base com sua respectiva altura.
- ➔ Calcula o valor absoluto da diferença entre o retângulo maior e a soma da área dos menores.
- ➔ Finalmente, compara com o erro que é a tolerância dada pelo usuário, se estiver dentro do limite tolerável, então a função retorna o valor da soma dos dois retângulos menores, caso contrário a função é chamada recursivamente, passando como argumento os parâmetros dos retângulos menores.

- **Concorrente**

As soluções concorrentes foram baseadas na sequencial, usando a mesma função quad_adaptativa() com apenas algumas pequenas adaptações. Primeiramente iremos falar sobre a solução balanceada com lista encadeada.

O nosso algoritmo para esta solução é:

- ➔ Pega o input do usuário.
- ➔ Põe o input do usuário em uma struct do tipo “parametro”, e da um push() nessa struct, na pilha. Nesse momento, o primeiro elemento da pilha é o input do usuário.
- ➔ Em seguida cria as threads, e chama a função calcula_thread(), que é a função com que todas as threads são chamadas quando são criadas.
- ➔ A função calcula_thread() vai retornar com pthread_exit(), a soma das áreas

aceitáveis das threads.

- ➔ Ela vai primeiro declarar o ponteiro ret, que será retornado para a main.
- ➔ Depois vai chamar a função de lock no mutex.
- ➔ Em seguida, vai entrar em um loop, que vai executar enquanto ainda tiver elemento na pilha, ou alguma thread ainda estiver no meio do cálculo de um retângulo. É necessário dar lock no mutex pois para avaliar a entrada no loop é preciso acessar a variável global que representa o topo da pilha, e a outra que indica se tem thread calculando. Ambas podem ter seu valor alterado por qualquer thread.
- ➔ Após entrar no loop, a thread vai pegar o próximo elemento na pilha, chamando a função pega_proximo_elemento(), passando como argumento a soma local total até o momento, e o ponteiro de retorno.
- ➔ A função pega_proximo_elemento(), vai entrar em um loop se a pilha estiver vazia mas tiver alguma thread calculando, neste caso a thread vai entrar em estado de espera, até que tenha um elemento na pilha para ser calculado, ou até que não tenha elemento na pilha, e nenhuma thread calculando, significando que acabaram os retângulos para serem calculados, neste caso, a função retorna alí mesmo com os argumentos passados pela thread.
- ➔ Se nenhuma dessas condições forem verdadeiras, então a thread sinaliza que vai começar a calcular incrementando a variável “calculando”, retorna o elemento no topo da pilha, da unlock no mutex, e adiciona em sua variável local o resultado da chamada da função quad_adaptativa() com os argumentos que foram retirados do topo da pilha.
- ➔ A função quad_adaptativa() funciona da mesma maneira que na versão sequencial, até a parte em que se compara o erro fornecido pelo usuário. Nessa parte, se o erro for aceitável:
    - ◆ Ela retorna para a função calcula_thread() a área do retângulo.
    - ◆ A função calcula_thread() pede o mutex, sinaliza que terminou de calcular decrementando a variável “calculando”.
    - ◆ Sinaliza com broadcast para as outras threads que terminou seu cálculo, para que elas chequem a pilha e o estado das outras threads.
    - ◆ E avalia se já pode sair do loop -se a pilha está vazia e não tem outra thread calculando- e retornar, ou se ainda continua a calcular, neste

caso chamando novamente a função pega_próximo_elemento() e continuando o ciclo.

- ➔ Caso contrário, declara duas structs, uma para cada um dos retângulos menores, pede o mutex, da push() em um dos retângulos, sinaliza a condição de que existe agora um elemento na pilha, devolve o mutex, e chama a função quad_adaptativa() novamente, para o outro retângulo, e fazendo assim até que possa retornar para a função calcula_thread() continuar sua execução.

O algoritmo para a solução concorrente desbalanceada funciona assim:

- ➔ A main recebe o input do usuário.
- ➔ Divide o intervalo [a, b] pelo número de threads e armazena em um vetor.
- ➔ Cria as threads.
- ➔ Cada thread vai ter seu próprio índice no vetor, por esse motivo não é necessário usar variáveis de sincronização nem de exclusão mútua.
- ➔ A função calcula_thread() dessa versão, aloca memória para o vetor de retorno, e chama a função quad_adaptativa(), para calcular seu intervalo.
- ➔ A função quad_adaptativa() por sua vez, é igual a sua versão na solução sequencial.
- ➔ Cada thread vai retornar o resultado de seu respectivo intervalo para a main, que vai fazer a soma deles e finalizar o programa.

O que fizemos nessa versão foi basicamente dividir o intervalo em tamanhos iguais, e distribuir para as threads. Ela é desbalanceada porque uma thread pode receber um intervalo que demora muito para calcular, e as outras threads podem terminar muito mais rápido que ela.

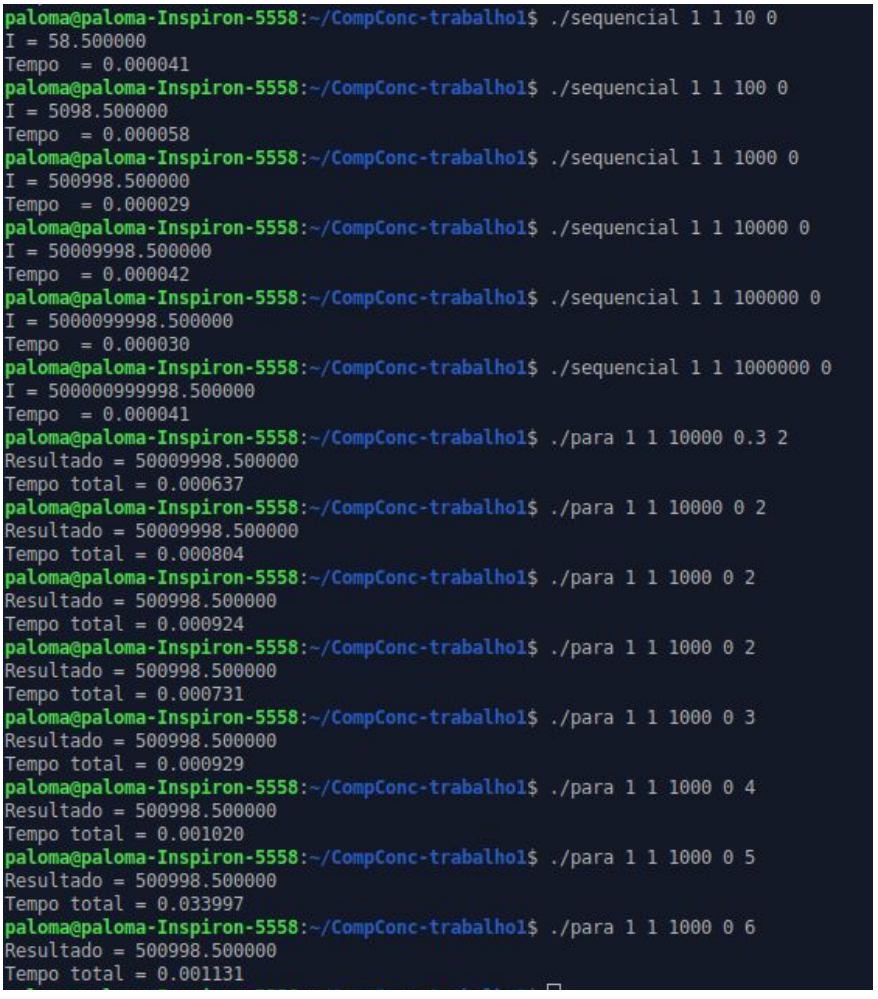
5. Testes

Para executar a função sequencial, o usuário deve fornecer, na linha de comando, nesta ordem: o número correspondente da função, o intervalo do lado esquerdo, o intervalo do lado direito, e o erro máximo.

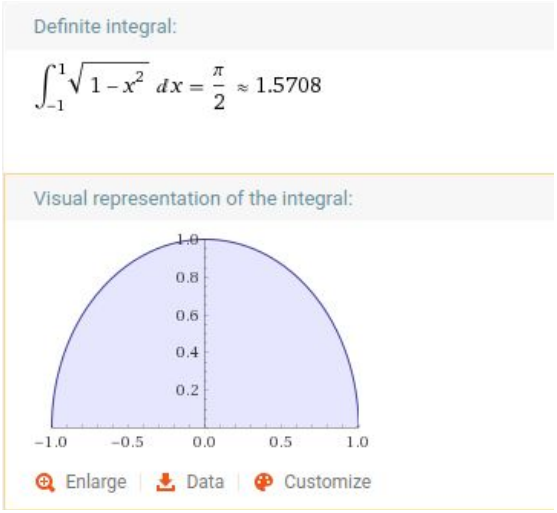
Para executar as funções paralelas, o usuário deve fornecer os mesmos argumentos fornecidos para a função sequencial, porém com um argumento extra no final, que é o número de threads.

Estabelecemos os parâmetros de teste de acordo com a ferramenta do WolframAlpha.

- $f_1 = x + 1$;
  Sabendo que a primeira função é linear e bem comportada no intervalo, o cálculo da integral pelo método tem um bom comportamento com o método as quadraturas adaptativas.
  Porém. na versão paralela desbalanceada este padrão não acompanha, o erro máximo não atinge zero e temos que interromper o processamento.
  Até 4 threads (o número de processadores da máquina) temos ganho de desempenho em comparação com mais threads e erro máximo, porém em comparação com a versão sequencial, não há ganho.



- $f(x) = \sqrt{1 - x^2}, \ -1 < x < 1$



Nesta função, a versão sequencial não teve um bom desempenho em comparação à paralela. De acordo com a ferramenta de cálculo de integral, o resultado é aproximadamente 1.5708.

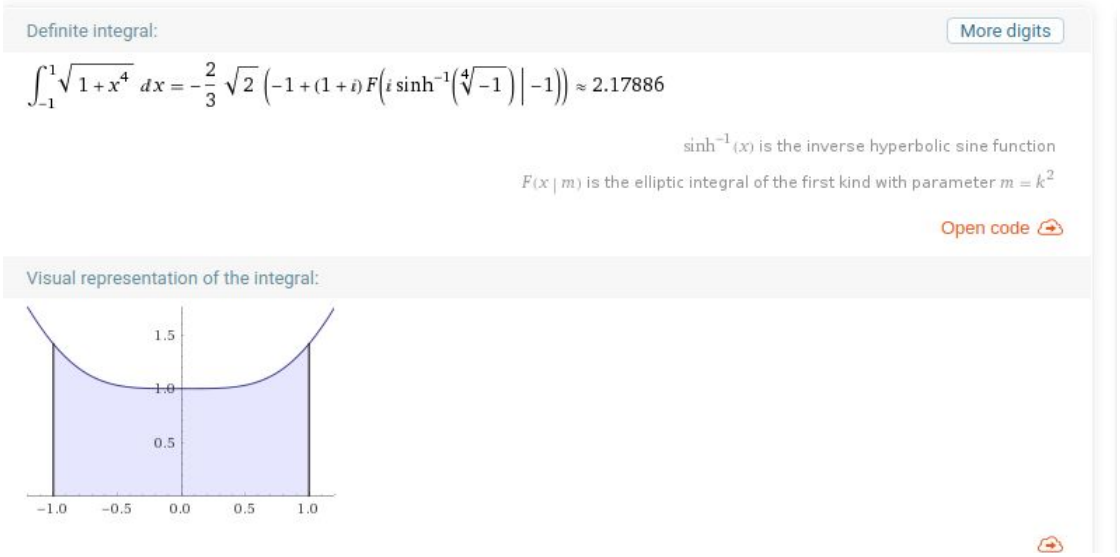
Como podemos ver não importa o quanto aumentamos o erro máximo que o valor da integral permanece a mesma. Ao executar a versão paralela, o valor

```
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 2 -1 1 10
I = 1.732051
Tempo  = 0.000041
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 2 -1 1 5
I = 1.732051
Tempo  = 0.000041
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 2 -1 1 4
I = 1.732051
Tempo  = 0.000052
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 2 -1 1 3
I = 1.732051
Tempo  = 0.000042
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 2 -1 1 2
I = 1.732051
Tempo  = 0.000028
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 2 -1 1 1
I = 1.732051
Tempo  = 0.000041
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 2 -1 1 0
x tem que estar entre -1 e 1
x tem que estar entre -1 e 1
x tem que estar entre -1 e 1
x tem que estar entre -1 e 1
^C
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 2 -1 1 100
I = 1.732051
Tempo  = 0.000041
```

Já a versão paralela conforme vamos aumentando o erro máximo o tempo de execução também aumenta e o mesmo acontece com o aumento do número de threads.

```
Tempo total = 0.001611
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 7 -4 4 1 2
Resultado = 4.733751
Tempo total = 0.000821
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 2 -1 1 1 2
Resultado = 1.732051
Tempo total = 0.000998
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 2 -1 1 1 3
Resultado = 1.732051
Tempo total = 0.000917
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 2 -1 1 0.5 3
Resultado = 1.732051
Tempo total = 0.000676
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 2 -1 1 0.5 2
Resultado = 1.732051
Tempo total = 0.000846
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$
```

- $f(x) = \sqrt{1+x^4}$



Adotamos o mesmo intervalo anterior para analisarmos o comportamento da integração quando a função cresce de maneira mais rápida. Desta vez as duas versões obtiveram o mesmo resultado, porá com tempos de execução diferentes. e o caso de aumento de tempo conforme aumentamos o erro máximo ou o número de threads, permaneceu. A versão sequencial ainda não tem ganho no processamento.

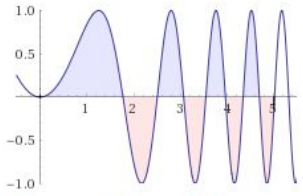
```
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 3 -1 1 1 2
Resultado = 2.061553
Tempo total = 0.000814
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 3 -1 1 1
I = 2.061553
Tempo  = 0.000065
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 3 -1 1 1 3
Resultado = 2.061553
Tempo total = 0.000891
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 3 -1 1 1
I = 2.061553
Tempo  = 0.000055
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 3 -1 1 1
```

- $f(x) = sin(x^2)$

Definite integral:

$$\int_0^5 \sin(x^2)\,dx = \sqrt{\frac{\pi}{2}}\, S\left(5\sqrt{\frac{2}{\pi}}\right) \approx 0.527917$$

Visual representation of the integral:



Enlarge | Data | Customize

Escolhemos o intervalo [0,5] para analisar o comportamento nas áreas menores. As duas versões obtiveram o mesmo resultado quando postos com o mesmo erro máximo. Inclusive ao erro tender a zero o resultado convergia com o esperado. E o mesmo comportamento se manteve, da piora do tempo e não obtivemos melhorias de desempenho com o paralelismo.

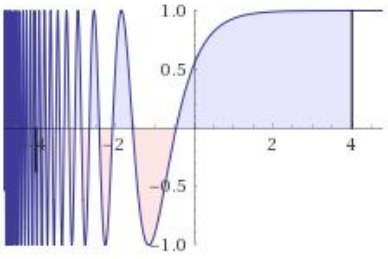
```
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 4 0 5 1
I = 0.084579
Tempo  = 0.000083
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 4 0 5 0
^C
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 4 0 5 1
I = 0.084579
Tempo  = 0.000072
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 4 0 5 2
I = 0.084579
Tempo  = 0.000100
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 4 0 5 0.5
I = 0.385435
Tempo  = 0.000100
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 4 0 5 0.05
I = 0.518390
Tempo  = 0.000084
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 4 0 5 0.005
I = 0.528201
Tempo  = 0.000151
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 4 0 5 0.003
I = 0.525238
Tempo  = 0.000172
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 4 0 5 1 2
Resultado = 0.084579
Tempo total = 0.000962
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 4 0 5 0.5 3
Resultado = 0.385435
Tempo total = 0.000787
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 4 0 5 0.5 2
Resultado = 0.385435
Tempo total = 0.000726
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 4 0 5 0.005 2
Resultado = 0.528201
Tempo total = 0.001019
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$
```

- $f(x) = cos(e^{-x})$

Definite integral:

$$\int_{-4}^{4} \cos(e^{-x})\,dx = \text{Ci}(e^4) - \text{Ci}\left(\frac{1}{e^4}\right) \approx 3.40599$$

Visual representation of the integral:



Indefinite integral:

Selecionamos o intervalo [-4,4]. Quando o erro tende a zero (nas duas versões) o valor se aproxima bastante com o esperado, porém na forma sequencial ainda é bem mais rápido.

```
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 5 -4 4 1 2
Resultado = 4.506118
Tempo total = 0.000805
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 5 -4 4 2 2
Resultado = 5.756850
Tempo total = 0.000876
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 5 -4 4 2 8
Resultado = 5.756850
Tempo total = 0.001779
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 5 -4 4 3 8
Resultado = 5.756850
Tempo total = 0.001256
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 5 -4 4 0.5 8
Resultado = 4.506118
Tempo total = 0.001714
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 5 -4 4 0.02 8
Resultado = 3.398145
Tempo total = 0.001789
```

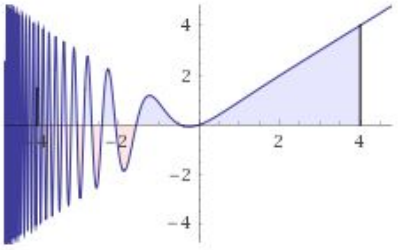
```
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 5 -4 4 0.02
I = 3.398145
Tempo  = 0.000198
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 5 -4 4 1
I = 4.506118
Tempo  = 0.000077
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 5 -4 4 0.5
I = 4.506118
Tempo  = 0.000093
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$
```

- $f(x) = cos(e^{-x}) * x$

Definite integral:

$$\int_{-4}^{4} x \cos(e^{-x})\, dx = 8.31266$$

Visual representation of the integral:

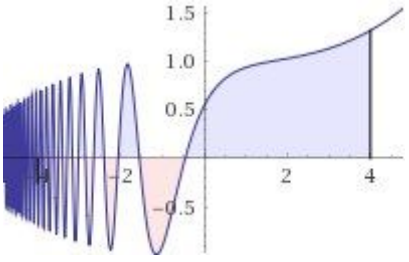


Seguindo o mesmo comportamento, inclusive, quando aumentado o número das threads, o tempo de execução aumenta.

```
Tempo  = 0.000093
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 6 -4 4 0.5 8
Resultado = 6.065332
Tempo total = 0.001402
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 6 -4 4 0.005 8
Resultado = 8.312692
Tempo total = 0.002104
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 6 -4 4 0.005 2
Resultado = 8.312692
Tempo total = 0.001098
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 6 -4 4 0.005 3
Resultado = 8.312692
Tempo total = 0.001311
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$
```

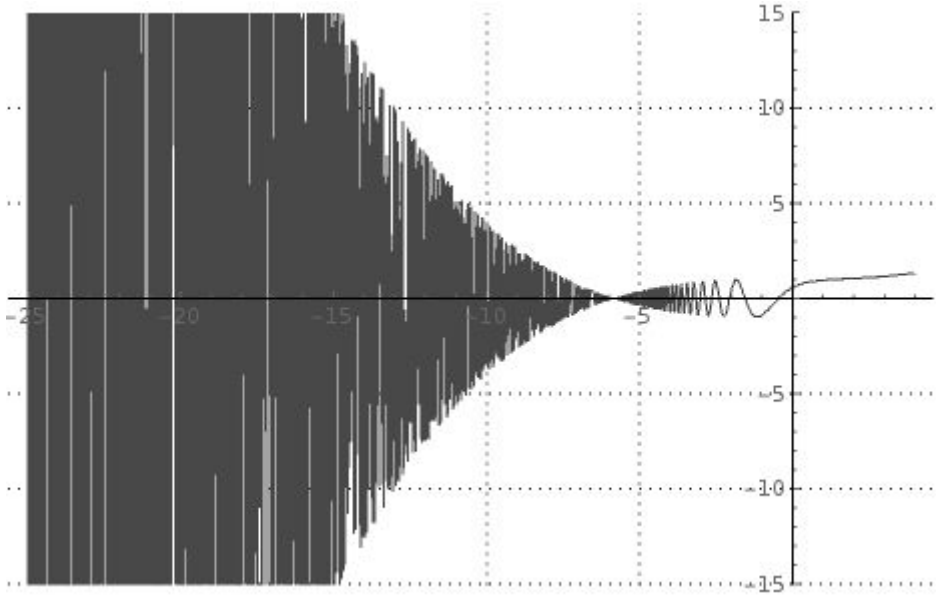
```
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 6 -4 4 0.02
I = 8.297883
Tempo  = 0.000214
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 6 -4 4 0.005
I = 8.312692
Tempo  = 0.000276
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 6 -4 4 2
I = 6.371920
Tempo  = 0.000040
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 6 -4 4 8
I = 4.339999
Tempo  = 0.000043
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 6 -4 4 0.75
I = 5.981237
Tempo  = 0.000035
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$
```

- $f(x) = cos(e^{-x}) * (0.005 * x^3 - 1)$



Neste caso os valores esperados e o valor obtidos foram aproximados, mas ambas versões mantiveram o comportamento das funções anteriores mediante a variação do erro máximo e do número de threads.

A função também possui um comportamento interessante do ponto x ≈ -6 à x tendendo a $-\infty$. Como mostrado na figura abaixo:



No intervalo [-6, -25], observamos que a solução paralela com array, é mais rápida que a solução sequencial - que executou em média com 7.3 a 7.5 segundos - , com o melhor desempenho atingido quando usamos 2 threads (média de 6 segundos) especificamente. Com apenas uma thread ela tem o desempenho um pouco inferior (8 segundos) do que a sequencial, com 3 fica um pouco menos eficiente (média de 7.5 a 7.8 segundos), e com 4 fica com desempenho praticamente igual a 3 threads. O erro usado nesse intervalo foi de 0.00001.

```
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 7 -4 4 0.5
I = 4.733751
Tempo  = 0.000079
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 7 -4 4 1
I = 4.733751
Tempo  = 0.000081
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 7 -4 4 2
I = 5.843650
Tempo  = 0.000079
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./sequencia 7 -4 4 0.000005
I = 3.731025
Tempo  = 0.001385
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 7 -4 4 0.00005 2
Resultado = 3.730989
Tempo total = 0.001346
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 7 -4 4 0.00005 3
Resultado = 3.730989
Tempo total = 0.001611
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$ ./para_array 7 -4 4 1 2
Resultado = 4.733751
Tempo total = 0.000821
paloma@paloma-Inspiron-5558:~/CompConc-trabalho1$
```

4. RESULTADOS

De acordo com os códigos executamos nos testes, notamos que a versão sequencial se manteve com melhor desempenho. porém quando o erro máximo tendia a zero, os tempos de execução eram bem próximos.

Na versão paralela caso o número de threads fossem maior que 2 o desempenho

se tornava pior do que com 2 threads, mesmo o computador possuindo 4 núcleos.

5.CONCLUSÃO

Acreditamos que um dos motivos principais da solução concorrente ter um desempenho pior que o da sequencial é o algoritmo faz muitos acessos à variáveis globais como o topo de pilha em todas as vezes que um método que faz operações na pilha é chamado, e consequentemente isso gera muitas seções críticas, e isso sabemos que é ruim para o desempenho. O preço que estamos pagando para usar as threads, então, não está valendo a pena nesse caso.

O mau desempenho da solução com lista encadeada nos levou a tentar uma abordagem com arrays, deixando de se preocupar com memória, e tentando ganhar um pouco de desempenho sem ter que ficar alocando memória o tempo todo. Esta abordagem não teve uma melhora significativa no desempenho, ficou apenas levemente mais rápida que a abordagem com lista encadeada. O algoritmo da solução com array é o mesmo que o algoritmo usando lista encadeada, porém as operações de push e pop são feitas num array de tamanho pré-definido.

Tentamos desenvolver um algoritmo que não fizesse tantos acessos a variáveis globais, que não precisasse de pedir tanto mutex, mas não conseguimos pensar em um que fosse também balanceado. Então por isso resolvemos implementar uma versão desbalanceada, para comparar com a outras versões e ver se haveria um ganho significativo. Depois de realizar os testes, descobrimos que no pior caso, a versão desbalanceada tinha o mesmo desempenho da versão sequencial, e em alguns casos havia ganho de desempenho expressivo, então no geral, essa versão tem um desempenho melhor, mas infelizmente como ela é desbalanceada, o ganho de desempenho dependerá da função que é passada como parâmetro então não é possível generalizar.

Sabendo que para todo algoritmo recursivo, existe um correspondente iterativo. Acreditamos que ao implementar o não recursivo na versão paralela, o ganho de desempenho aumentaria expressivamente em comparação com o atual cenário. Pois quase sempre consomem mais recursos, como memória, por conta do uso da pilha, como já citado.