Relatório para o trabalho prático da disciplina Computação Concorrente

Matheus Fernandes[1], **Stephanie Orazem**[1]

[1] Departamento de Ciência da Computação
Universidade Federal do Rio de Janeiro (UFRJ) – Rio de Janeiro, RJ – Brazil

Link pro repositório no Github

matheusfcc2010@hotmail.com, sorazemhr@gmail.com

1. Introdução

Neste trabalho nos foi proposto realizar a implementação sequencial e concorrente do método de integração numérica retangular com estratégia de quadratura adaptativa. Esta se trata de aproximar o valor de uma integral dado um intervalo [a, b]. Calcula-se o ponto médio m entre a e b, o valor da função nesse ponto e com isso a área do retângulo de altura $f(m)$ e largura $b - a$. Depois calculamos pontos médios nos intervalos [a, m] e [m, b], seus valores na função e as áreas dos novos retângulos formados.

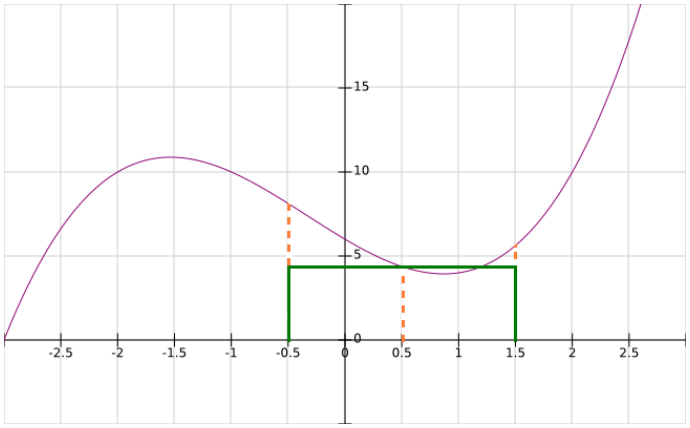


Figura 1. Área do retângulo maior.

Dado um erro máximo da aproximação da integral, observamos a diferença entre a área do retângulo maior e a soma dos retângulos menores e comparamos com o erro. Se o módulo de tal diferença for menor, então assumimos que a integral do intervalo [a, b] é a área do retângulo maior. Caso contrário, repetimos o procedimento com os subintervalos gerados até todos a solução de cada subproblema convergir.

2. Implementação

Para seguir o método do problema utilizamos uma estrutura de dados, no caso uma **pilha**, para guardar os intervalos, sendo que primeiramente ela guarda somente o intervalo inicial recebido na entrada e, conforme o programa vai fazendo os cálculos descritos acima, ele vai tirando e colocando novos intervalos nesta pilha enquanto não atingir um valor menor que o erro dado. O ponto de parada acontece somente quando a pilha não tiver mais nenhum intervalo para calcular.

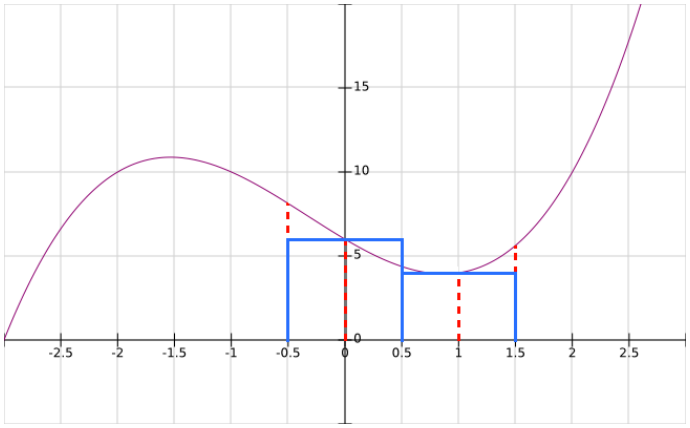


Figura 2. Área dos retângulos menores.

Seguimos por esse caminho pois num momento inicial evitamos o uso de recursividade, que era uma possível solução para o problema, entretanto poderia gastar muita memória da máquina, principalmente na versão concorrente, sem contar a dificuldade de desenvolver a mesma nesta versão. Inspirados pela disciplina *Algoritmos e Grafos* chegamos nessa estratégia implementada.

2.1. Estruturas de Dados

Para guardar apropriadamente os dados do problema, tanto da entrada quanto das variáveis internas, criamos uma *struct* chamada **Intervalos** que guarda o início e fim do intervalo, a e b respectivamente, e outra *struct* chamada **Pilha** que possui tamanho, posição do último elemento adicionado nela e um vetor de **Intervalos**, além dos métodos conhecidos dessa estrutura que são *push, pop, peek, isEmpty, isFull* e *init*. Nestes passamos a pilha por referência.

2.2. Abordagem Sequencial

A partir da entrada do problema, obtemos o intervalo inicial e o erro máximo. Transformamos o intervalo inicial na nossa estrutura Intervalos e colocamos na pilha de Intervalos. Após isso, começamos a calcular a integral: avaliamos e tiramos o intervalo do topo da pilha e seguimos os passos apresentados na introdução deste relatório, sendo que se o módulo da diferença calculado for menor que o erro estipulado então adicionamos no valor da integral a área do retângulo maior, senão chamamos o método *push* para introduzirmos os subintervalos à pilha. Repetimos esses passos até chegar no critério de parada citado anteriormente.

2.3. Abordagem Concorrente

A versão concorrente do problema parte do mesmo princípio da abordagem sequencial, ou seja, ela faz uso das mesmas estruturas de dados e do mesmo fluxo de lógica. Cada *thread* será criada para realizar os passos descritos na versão sequencial. Como a pilha pode ser acessada por mais de uma *thread*, fizemos uso da técnica de exclusão mútua com *locks*. Além disso, analisamos outros casos críticos para o programa:

- Pilha vazia

Consideramos o caso onde uma *thread* irá analisar o topo da pilha e irá verificar que a pilha está vazia. Teoricamente, nessa situação, a *thread* deveria parar de calcular a integral, entretanto, é possível que uma outra *thread* esteja ainda analisando um outro intervalo e irá colocar novos intervalos na pilha. Isto indica que o cálculo da integral não deve ser finalizado ainda. Portanto, para evitar este erro, bloqueamos a *thread* no momento em que ela verificou que a pilha está vazia. Existem dois modos para a *thread* ser liberada:
Quando uma outra *thread* adicionar novos intervalos na pilha, esta irá enviar um sinal (*broadcast*). E quando a última *thread* não produzir mais subintervalos, ela irá verificar que a pilha está vazia e desbloquear todas as outras *threads* que estão no aguardo. Para isto, mantemos uma variável que guarda quantas *threads* estão esperando para ser liberadas.

- Pilha cheia

  Consideramos agora o caso onde uma *thread* irá adicionar novos intervalos na pilha, mas irá verificar que a pilha está cheia. Naturalmente, ela não poderá realizar a ação desejada e irá se bloquear até receber um sinal de uma *thread* que pegará o próximo intervalo do topo da pilha, liberando espaço.

2.4. Problemas

Enquanto desenvolvemos o código do trabalho, encontramos algumas barreiras. A primeira delas foi a implementação da estrutura de dados utilizada no programa, a pilha, mais especificamente na expansão da mesma. Queríamos que toda vez que ela fosse totalmente preenchida ela dobrasse a sua capacidade, entretanto não conseguimos fazer por um problema de passagem por referência do ponteiro da pilha, ela conseguia alocar a memória dobrada mas a pilha do programa não recebia o endereço de onde foi alocada. Tentamos solucionar, mas percebemos que estávamos gastando um tempo precioso e então optamos por trabalhar a pilha com tamanho fixo.

Uma outra barreira que encontramos durante o desenvolvimento foi um problema de condição de erro da aplicação concorrente. Detectamos um fluxo de execução errado que o programa poderia seguir, que seria uma *thread*, após encontrar a diferença das áreas de sua iteração sendo maior que o erro e ter que colocar novos intervalos na pilha, observa que a pilha está totalmente ocupada e portanto deve se bloquear e esperar abrir um espaço para poder inserir. Entretanto, por algum motivo, o programa insere mais do que retira espaços da pilha e chega em um momento que ele entra em *deadlock*, ou seja, todas as *threads* ficam bloqueadas, apesar de que em nossa visão a lógica do tratamento dessa condição está implementado corretamente. Não conseguimos pensar em uma alternativa para solucionar esse problema e por isso nosso programa possui um limite dependendo do espaço de pilha utilizado, da quantidade necessária de retângulos pra calcular a integral e do número de *threads* trabalhando. Se tiver pouco espaço na pilha, precisar de uma grande quantidade de retângulos pra calcular-se a integral e for dado um número razoável de *threads* para trabalhar, acima de 4 por exemplo, há grandes chances do programa entrar em *deadlock*.

Também sentimos dificuldade em propor uma solução explícita para o balanceamento de carga, com variáveis auxiliares e funções que pudessem realizar esse trabalho.

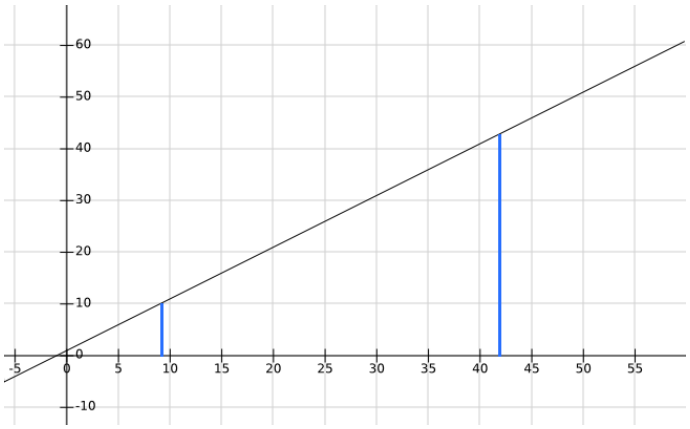
Como as *threads* podem se bloquear durante os cálculos, então não poderíamos utilizar a solução feita em sala de aula pois o programa certamente iria entrar em *deadlock*. Entretanto, como as *threads* fazem uma quantidade de cálculos bastante próximos uns dos outros, de vez em quando há uma grande diferença entre as cargas de trabalho, então não sentimos a necessidade de propor o balanceamento.

3. Casos de Teste

Neste tópico iremos mostrar os resultados do programa para as funções citadas na descrição do trabalho e nossas conclusões a partir deles.

Utilizamos em todos os nossos testes o erro 10^{-15} para que o programa tenha um tempo de execução um pouco mais demorado e a gente possa ter uma análise melhor do que esteja se passando nele. Além disto, para a análise, mostramos na saída o número de retângulos que o programa usou para calcular a integral, que em na maioria dos casos de execução sequencial e concorrente sempre é utilizado a mesma quantidade, quando não a diferença é de pouquíssimos; e o tempo para realizar a mesma.

(a) $f(x) = x + 1$

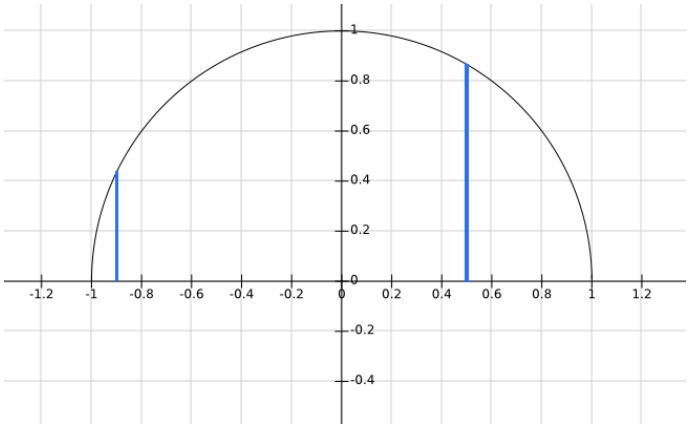


Intervalo usado: [9, 42]

Valor real aproximado da integral no intervalo acima: 874.5
Valor obtido aproximado pelo programa sequencial: 874.5
Valor obtido aproximado pelo programa concorrente: 874.5

Neste exemplo, como a área debaixo da função será um trapézio, um polígono simples para se calcular tal, então os algoritmos sequencial e concorrente conseguem obter o resultado rapidamente, sem apresentar uma grande diferença entre seus tempos de execução. Além disso, ambos utilizam somente um retângulo para calcular a integral, o que reforça esse ponto.

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



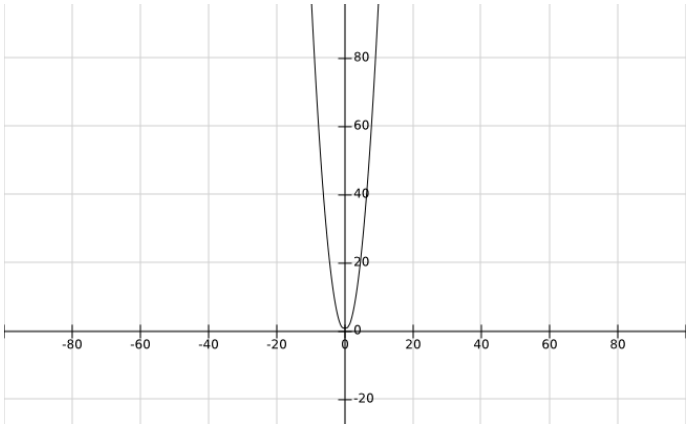
Intervalo usado: [-0.9, 0.5]

Valor real aproximado da integral no intervalo acima: 1.23434
Valor obtido aproximado pelo programa sequencial: 1.23434
Valor obtido aproximado pelo programa concorrente: 1.23434

Número de retângulos utilizados para processar a integral: 73.037

Neste exemplo ambas as execuções sequencial e concorrente também resolvem a integral em tempos muito pequenos. Acreditamos que seja por causa da função ser bem comportada, ou seja, não possui oscilações, estar definida em um intervalo pequeno e também por ela não possuir valores muito grandes, apesar de ter sido utilizado bastante retângulos para obter-se o resultado.

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



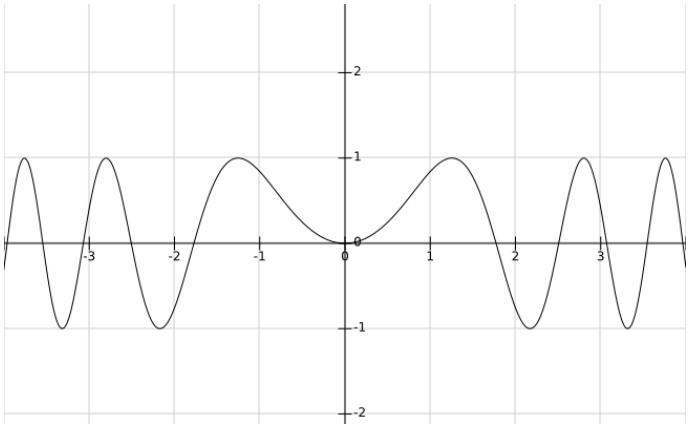
Intervalo usado: [15, 200]
Valor real aproximado da integral no intervalo acima: $2.66554169749 * 10^6$
Valor obtido aproximado pelo programa sequencial: $2.66554169750 * 10^6$
Valor obtido aproximado pelo programa concorrente: $2.66554169750 * 10^6$

Número de retângulos utilizados para processar a integral: 8.388.608

Tempo médio de execução na versão sequencial: 7s
Tempo médio de execução na versão concorrente com 2 *threads*: 10s
Tempo médio de execução na versão concorrente com 3 *threads*: 11s
Tempo médio de execução na versão concorrente com 4 *threads*: 13s

Percebemos que agora temos um tempo de execução diferente dos anteriores, porém a versão sequencial é mais veloz e acreditamos que seja pela função crescer para valores altíssimos muito rapidamente.

(d) $f(x) = sen(x^2)$



Intervalo usado: [-30, 30]

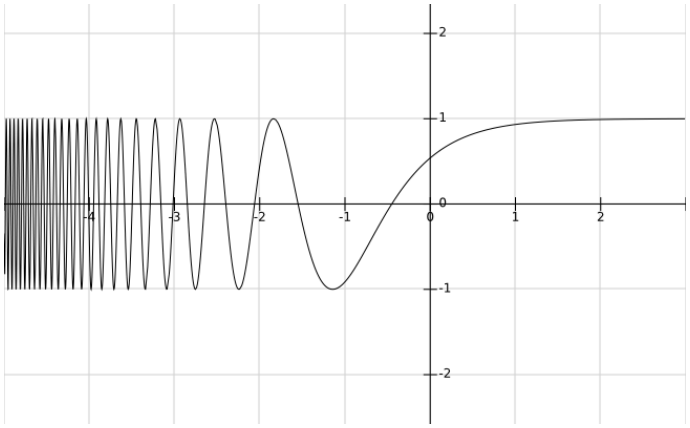
Valor real aproximado da integral no intervalo acima: 1.25108
Valor obtido aproximado pelo programa sequencial: 1.25109
Valor obtido aproximado pelo programa concorrente: 1.27963

Número de retângulos utilizados para processar a integral: 20.816.460

Tempo médio de execução na versão sequencial: 28s
Tempo médio de execução na versão concorrente com 2 *threads*: 22s
Tempo médio de execução na versão concorrente com 3 *threads*: 31s
Tempo médio de execução na versão concorrente com 4 *threads*: 33s

Vemos que desta vez a versão concorrente que começa a ser um pouco mais veloz que a sequencial, muito provavelmente por causa do comportamento da função, ela é simétrica, não possui valores muito altos e oscila regularmente. Notamos que para intervalos menores, a execução concorrente com 2 *threads* era bem próxima da sequencial e ambas eram distantes das concorrentes com mais *threads*, mas conforme aumentamos o intervalo de integração a versão sequencial se distanciava da versão com 2 *threads*, então é válido supor que para valores maiores que os testados a versão concorrente com mais threads poderá ser mais rápida.

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



Intervalo usado: [-9, 3]

Valor real aproximado da integral no intervalo acima: 2.42330
Valor obtido aproximado pelo programa sequencial: 2.42331
Valor obtido aproximado pelo programa concorrente: 2.42331

Número de retângulos utilizados para processar a integral: 22.459.199

Tempo médio de execução na versão sequencial: 26s
Tempo médio de execução na versão concorrente com 2 *threads*: 22s
Tempo médio de execução na versão concorrente com 3 *threads*: 32s
Tempo médio de execução na versão concorrente com 4 *threads*: 34s

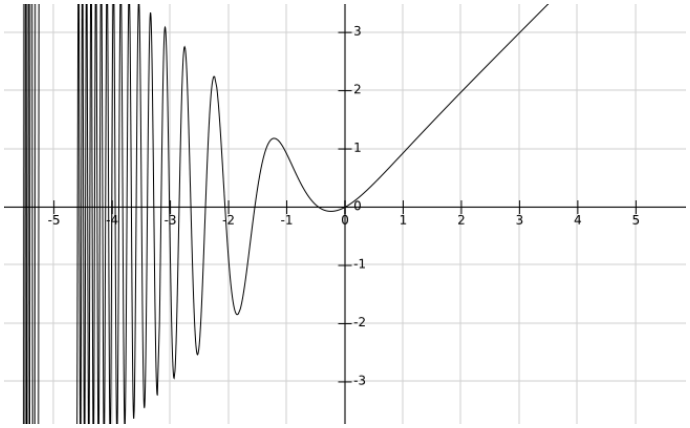
Observamos que este se comporta de forma similar com o exemplo anterior, o programa sequencial e concorrente com 2 *threads* nos intervalos menores gastam aproximadamente o mesmo tempo e menores do que no concorrente com mais *threads*. Entretanto, existe algo mais característico nesta função: para valores negativos ela oscila bastante e para valores positivos isto não ocorre, isso impacta o programa de modo que se analisado um intervalo com maior parte negativa, o tempo da versão concorrente tende a ser menor que o da sequencial, enquanto se analisado um intervalo com maior positiva acontece o contrário.

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

Intervalo usado: [-6, 0]

Valor real aproximado da integral no intervalo acima: 0.35273
Valor obtido aproximado pelo programa sequencial: 0.35274
Valor obtido aproximado pelo programa concorrente: 0.35274

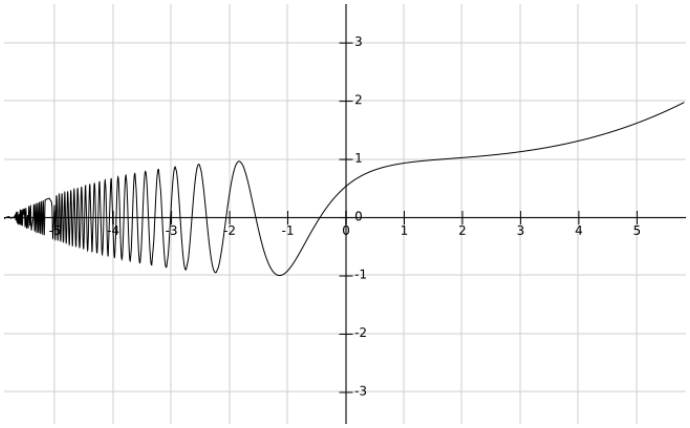
Número de retângulos utilizados para processar a integral: 4.938.938



Tempo médio de execução na versão sequencial: 6s
Tempo médio de execução na versão concorrente com 2 *threads*: 4s
Tempo médio de execução na versão concorrente com 3 *threads*: 7s
Tempo médio de execução na versão concorrente com 4 *threads*: 8s

Este exemplo se comporta de forma similar com o anterior, a principal diferença é que a parte oscilatória ondula com valores maiores.

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



Intervalo usado: [-8, 5]

Valor real aproximado da integral no intervalo acima: 5.20338
Valor obtido aproximado pelo programa sequencial: 5.20339
Valor obtido aproximado pelo programa concorrente: 5.20339

Número de retângulos utilizados para processar a integral: 10.277.683

Tempo médio de execução na versão sequencial: 16s
Tempo médio de execução na versão concorrente com 2 *threads*: 11s
Tempo médio de execução na versão concorrente com 3 *threads*: 17s

Tempo médio de execução na versão concorrente com 4 *threads*: 17s

Novamente, este caso mostra resultados parecidos com a letra (e) e (f), o que não é uma surpresa se olharmos para os gráficos dessas funções. Todavia, no atual exemplo a versão concorrente tende a ser mais rápida pela parte oscilante não possuir valores muito grandes e ter ondulações mais ”frequentes”, o comprimento de onda é menor.

3.1. Conclusão

No geral, percebemos que a nossa solução apresenta resultados bem comportados, sempre retorna valores bem aproximados em relação ao erro dado e próximas o suficiente dos valores reais.

4. Avaliação de Desempenho

Iremos avaliar se conseguimos obter algum ganho de velocidade de execução com a versão concorrente em nossos testes. Para isso usaremos a fórmula $T(sequencial)/T(concorrente)$.

No caso (c), a versão sequencial foi executada em 7s e a concorrente com 4 *threads* em 13s. Neste caso, a versão concorrente possui uma perda de execução de aproximadamente 0,54. Podemos observar também que quanto maior o número de *threads*, maior o tempo de execução. Ou seja, usar *threads* acaba tendo um custo bem maior nesse caso.

No caso de teste (d), há um ganho de 1.27 com 2 *threads*. Mas com 4 *threads* temos uma perda de 0.84.

No caso de teste (e), obtemos ganho de execução de 1.18 com 2 *threads* e uma perda de 0.76 com 4 *threads*.

No teste (f), há um ganho de 1.5 com 2 *threads* e uma perda de 0.75 com 4 *threads*.

Por último, no teste (g) vemos um ganho de 1.45 com com 2 *threads* e uma perda de 0.94 com 2 e 3 com 2 *threads*.

Com isto concluímos que o ganho de execução depende não só da forma como implementamos o programa, mas também da equação de entrada. Percebemos então que onde há mais ondulações a versão concorrente tende a ter um desempenho melhor. Por outro lado, a versão sequencial é vitoriosa quando a função não possui muitas sinuosidades.

Já em relação à quantidade de retângulos usados para o cálculo da integral, nós esperávamos que a versão concorrente seria mais veloz, pois estaria calculando mais retângulos ao mesmo tempo. Porém isso não ocorre, como podemos verificar no caso (c), onde há muitos retângulos e a versão sequencial é mais rápida. Dito isto, não conseguimos concluir como a quantidade de retângulos impacta no desempenho do programa.