

Universidade Federal do Rio de Janeiro
Bacharelado em Ciência da Computação
Computação Concorrente



UFRJ

Computação Concorrente - 2019.2
Relatório de Trabalho Prático

João Ricardo Campbell Maia - 116111909
Thiago de Oliveira Silva - 111466197

2019

Introdução:

A vertente da computação dita Computação Concorrente busca definir um conceito onde atividades são executadas de forma simultânea (concorrentemente), sendo utilizada atualmente em diversas aplicações e em diferentes áreas.

No contexto da disciplina, se desenvolve entre os conceitos de *threads* e *processos*, estudando os impactos de programas que possuem mais de um fluxo de execução e comparando sua eficiência com versões sequenciais dos mesmos. Tem por objetivo também explorar ferramentas que possibilitem a implementação dos mesmos, otimizando assim o uso do processador e melhor aproveitamento dos recursos da máquina.

Objetivo:

A atividade consiste na implementação algorítmica de uma programa que calcule áreas através de **integrais definidas** utilizando o método de **integração numérica retangular com ponto médio**. São propostas versões **sequenciais** e **concorrentes** (*com balanceamento de carga entre as threads*) do mesmo programa e a comparação entre os programas revelando o ganho de desempenho obtido.

Desenvolvimento:

Foram utilizadas as ferramentas Geany (IDE), Sublime Text (Editor) e linguagem de programação C++ em ambiente Linux e Windows.

A solução apresentada consiste em construir um retângulo com largura entre os intervalos de integração da função e altura correspondente ao ponto de máximo. Definir um ponto médio na função e a partir daí gerar dois novos retângulos. Um do intervalo inicial até o ponto médio, outro do ponto médio ao intervalo final. Se a soma desses retângulos for muito maior do que o retângulo inicial, o processo se repete recursivamente nos dois retângulos. Foi criada também uma versão concorrente sem balanceamento para que pudesse ser comparada com a balanceada.

A seguir, uma breve descrição das versões com as soluções encontradas:

→ Versão sequencial:

Para tal, foram utilizados os seguintes mecanismos:

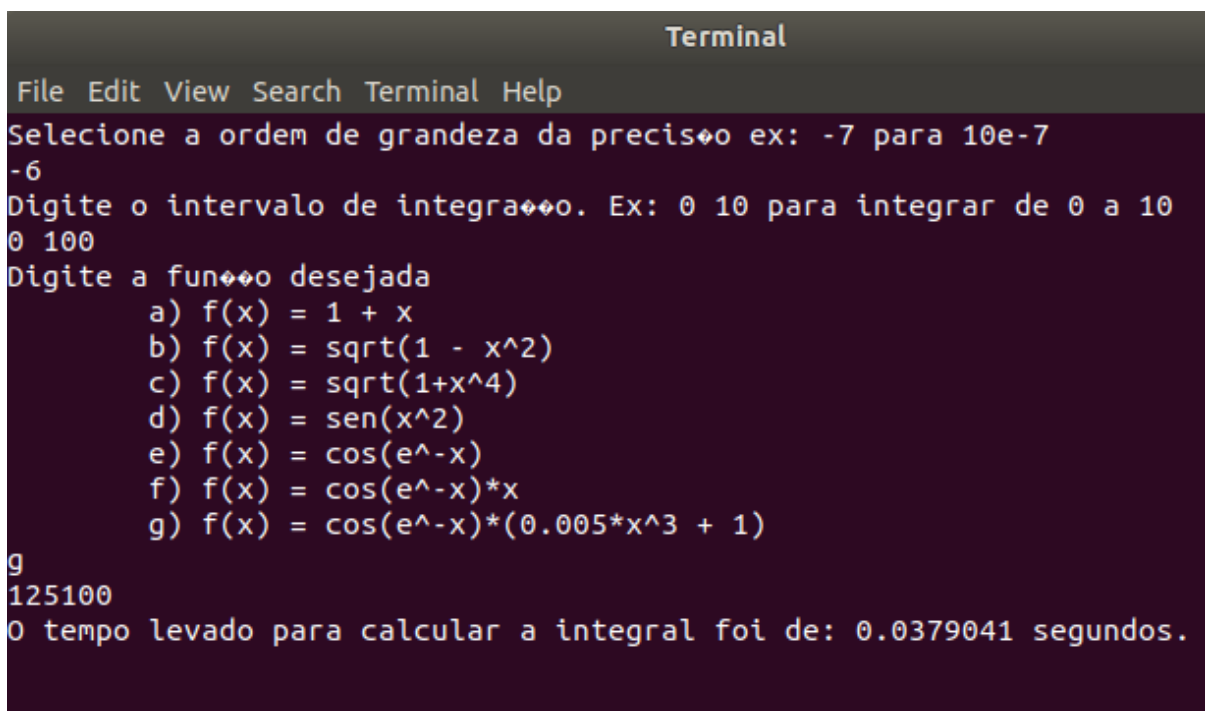
- Função que calcula o ponto médio
- Função que retorna o módulo
- Struct para representar os retângulos
- Função para o cálculo da integral
- Eventuais variáveis auxiliares como para definir a margem de erro

Fluxo de execução do programa

Ao executar, o programa fornece ao usuário a opção de escolher:

1. A ordem de grandeza da precisão
2. O intervalo de integração
3. A função de integração

O programa chama a função *Integral* que cria os retângulos e compara, até que a diferença entre o maior e a soma dos menores seja menor ou igual a margem de erro estipulada. A função executa recursivamente. O valor é acumulado em uma variável soma que é retornado para o usuário como resultado esperado. O programa mede também o tempo gasto na execução da tarefa. Um exemplo é mostrado a seguir (Figura 1).



```
Terminal
File Edit View Search Terminal Help
Selecione a ordem de grandeza da precisão ex: -7 para 10e-7
-6
Digite o intervalo de integração. Ex: 0 10 para integrar de 0 a 10
0 100
Digite a função desejada
a) f(x) = 1 + x
b) f(x) = sqrt(1 - x^2)
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)
g
125100
O tempo levado para calcular a integral foi de: 0.0379041 segundos.
```

Figura 1. Execução do programa sequencial

→ Versão concorrente desbalanceada:

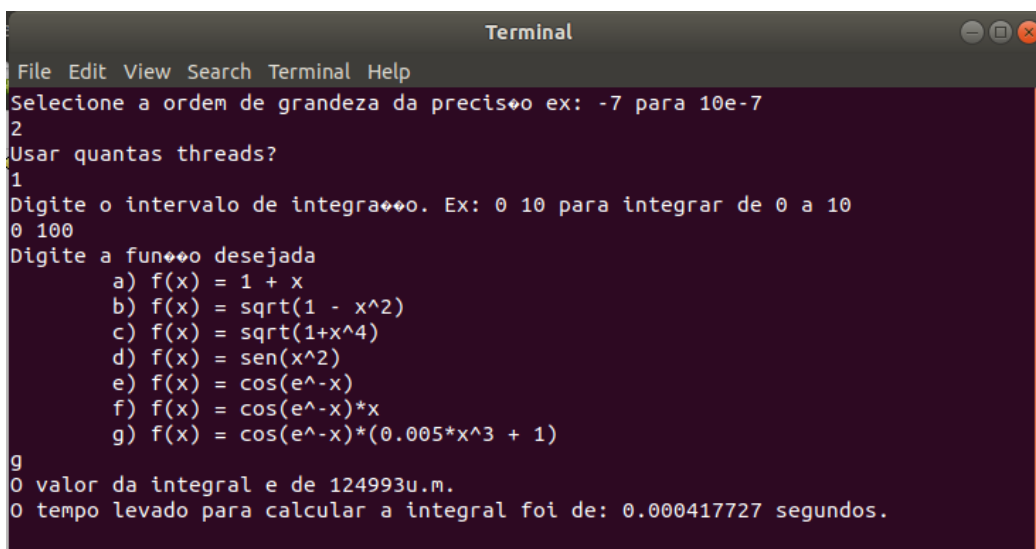
Nessa versão, foram adotadas as seguintes estratégias

- Um mutex da biblioteca Pthread para exclusão mútua
- Função que calcula o ponto médio
- Função que retorna o módulo
- Struct para representar os retângulos
- Função para o cálculo da integral
- Função IntegralConc (cria threads e divide a tarefa em blocos)
- Função IntegralAux (função para as threads)
- Struct para ser passada como argumento para a função IntegralAux
- Eventuais variáveis auxiliares

Fluxo de execução do programa

O programa inicia realizando os mesmos procedimentos da forma sequencial solicitando entradas do usuário.

*A função **IntegralConc** agora cria as threads e divide a tarefa em blocos. uma vez criadas, as threads executam a função **IntegralAux** de forma concorrente, onde cada thread, executa a função **Integral**, que antes era realizada por um único fluxo de execução, de forma sequencial. A função trata das devidas exclusões mútuas para que não haja acessos indevidos simultâneos na variável soma a ser incrementada. Para tal utiliza-se de locks. Ao final, a função IntegralConc utiliza-se de um “join” para aguardar todas as threads terminarem e imprimir o resultado. Um exemplo é mostrado a seguir (Figura2).*



```
Terminal
File Edit View Search Terminal Help
Selecione a ordem de grandeza da precisão ex: -7 para 10e-7
2
Usar quantas threads?
1
Digite o intervalo de integração. Ex: 0 10 para integrar de 0 a 10
0 100
Digite a função desejada
a) f(x) = 1 + x
b) f(x) = sqrt(1 - x^2)
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)
g
O valor da integral e de 124993u.m.
O tempo levado para calcular a integral foi de: 0.000417727 segundos.
```

Figura 2. Execução do programa concorrente desbalanceado

→ Versão concorrente balanceada

Para solucionar o problema do balanceamento de carga entre as threads, foi criado um **buffer**, que funciona como uma pilha de retângulos, implementada usando uma Struct. Temos agora também **dois mutex** (um para **soma** e outro para o **buffer**)

Fluxo de execução do programa

O programa inicia realizando os mesmos procedimentos da forma sequencial e concorrente desbalanceada solicitando entradas do usuário. A função **IntegralConc** agora, empilha no buffer os retângulos iniciais correspondentes ao número de threads sendo utilizadas. Já a função **IntegralAux** (realizada pelas threads) agora funciona como produtora e consumidora, se ela calculou um retângulo preciso, ela adiciona esse valor a sua soma, caso não, ela empilha mais dois retângulos no buffer para serem processados, até a pilha ficar vazia sinalizando que a todos os retângulos foram consumidos e a integral está completa. Um exemplo é mostrado a seguir (Figura 3).

```
Selecione a ordem de grandeza da precisão ex: -7 para 10e-7
-5
Usar quantas threads?
10
Digite o intervalo de integração. Ex: 0 10 para integrar de 0 a 10
0 1000
Digite a função desejada
a) f(x) = 1 + x
b) f(x) = sqrt(1 - x^2)
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)
d
Retângulo inicial de 0 a 100 acrescentado ao buffer.
Retângulo inicial de 100 a 200 acrescentado ao buffer.
Retângulo inicial de 200 a 300 acrescentado ao buffer.
Retângulo inicial de 300 a 400 acrescentado ao buffer.
Retângulo inicial de 400 a 500 acrescentado ao buffer.
Retângulo inicial de 500 a 600 acrescentado ao buffer.
Retângulo inicial de 600 a 700 acrescentado ao buffer.
Retângulo inicial de 700 a 800 acrescentado ao buffer.
Retângulo inicial de 800 a 900 acrescentado ao buffer.
Retângulo inicial de 900 a 1000 acrescentado ao buffer.
O valor da integral é de 1.51803 u.m.
O tempo levado para calcular a integral foi de: 3.04381 segundos.
-----
Process exited after 8.544 seconds with return value 0
```

Figura 3. Execução do programa concorrente balanceado com buffer

→ Versão concorrente balanceada c/ múltiplos buffers

Ao terminarmos a versão concorrente balanceada, notamos que esse acessos ao buffer com exclusão mútua deixavam a programa mais lento até que o sequencial em alguns casos, então resolvemos criar mais um método, onde se usam múltiplos buffers para os mutexes não atrasarem o programa demais, para isso, foram necessários os seguintes artefatos:

- Array de mutexes para os buffers
- Função RoubaRetangulos
- Mutex referente a Função Rouba Retangulos
- Número de Threads Ativas

O Array de Mutexes, que no caso tem tamanho oito, foi criado para reduzir a frequencia de acesso a um buffer individual significativamente, quase como uma tabela hash.

A função RoubaRetangulos serve para manter o balanceamento do programa, caso uma thread termine, ela busca em outros buffers um retangulo para processar. Um adendo importante é que a função RoubaRetangulos precisa de um mutex pois ela só pode estar sendo executada por uma thread, caso contrário, o programa pode entrar em deadlock.

O número de threads ativas serve para a thread saber em qual buffer ela estará, como temos oito buffers, usamos esse número mod 8 para identificar o seu buffer.

Fluxo de execução do programa

O programa começa da mesma forma que a balanceada, com a função **IntegralConc** empilhando um retangulo para cada thread em seu respectivo buffer e então as threads são criadas com a função **IntegralAux**, elas processam seus retangulos até seus buffers ficarem vazios, e então elas buscam em outros buffers com a função **RoubaRetangulos** para não ficarem sem processar nada, garantindo o balanceamento, assim que todos os buffers ficarem vazios, sabemos que a integração numérica terminou. Um exemplo de é mostrado a seguir (Figura 4).

```
Digite o intervalo de integratπo. Ex: 0 10 para integrar de 0 a 10
-6 10
Digite a funçπo desejada
a) f(x) = 1 + x
b) f(x) = sqrt(1 - x^2)
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)
g
Thread 8 Roubou o retangulo do buffer 1
Thread 8 Roubou o retangulo do buffer 1
Thread 8 Roubou o retangulo do buffer 1
Thread 8 Roubou o retangulo do buffer 1
Thread 8 Roubou o retangulo do buffer 1
Thread 8 Roubou o retangulo do buffer 1
O valor da integral e de 21.9214 u.m.
Calculado em 0.015604 segundos.
```

Figura 4. Execução do programa concorrente balanceado com múltiplos buffers

Resultados:

Abaixo segue uma planilha dos casos de teste e seus resultados (Figura 5). Ela pode ser conferida e modificada através do arquivo **Resultados.xlsx**.

Casos de Teste				Sequencial		Concorrente Desbalanceado		Concorrente Balanceado		Concorrente Balanceado c/ Múltiplos Buffers				Menor Tempo															
Função	Início	Fim	Precisão(s)	N_THREADS	Resultado	Tempo(s)	N_THREADS	Resultado	Tempo(s)	N_THREADS	Resultado	Tempo(s)	N_THREADS		Resultado	Tempo(s)													
a(x)	-10000	10000	1.00E-05	1	20000	0	8	20000	0.003	8	20000	0.000999	8	20000	0.002997	0													
a(x)	-1.00E+06	1.00E+06	1.00E-05	1	2.00E+06	0	8	2.00E+06	0.001989	8	2.00E+06	0.000998	8	2.00E+06	0.001998	0													
a(x)	-1.00E+06	1.00E+06	1.00E-09	1	2.00E+06	0	8	2.00E+06	0.001997	8	2.00E+06	0.00101	8	2.00E+06	0.002997	0													
b(x)	-1	1	1.00E-07	1	1.5708	0.000998	8	1.5708	0.000999	8	1.5708	0.001998	8	1.5708	0.000999	0.000998													
b(x)	-1	1	1.00E-03	1	1.57275	0	8	1.57275	0.002992	8	1.57218	0.002997	8	1.57275	0.000998	0													
b(x)	-1	0	1.00E-09	1	0.785398	0.000999	8	0.785398	0.000998	8	0.785398	0.000999	8	0.785398	0.002001	0.000998													
c(x)	-100	100	1.00E-05	1	666669	0.005999	8	666669	0.002922	8	666669	0.005996	8	666669	0.005996	0.002922													
c(x)	-10000	10000	1.00E-05	1	6.67E+11	0.787528	8	6.67E+11	0.174893	8	6.67E+11	1.4048	8	6.67E+11	0.378767	0.174893													
c(x)	-10000	10000	1.00E-07	1	6.67E+11	3.698718	8	6.67E+11	0.69857	8	6.67E+11	1.50507	8	6.67E+11	0.69857	0.69857													
d(x)	-100	100	1.00E-05	1	1.26292	0.118929	8	1.26292	0.040974	8	1.26292	0.14069	8	1.26292	0.07795	0.040974													
d(x)	-100	0	1.00E-07	1	0.631417	0.272833	8	0.631417	0.086946	8	0.631417	0.343356	8	0.631417	0.136915	0.086946													
d(x)	-100	100	1.00E-07	1	1.26283	0.539669	8	1.26283	0.180046	8	1.26283	0.627613	8	1.26283	0.341978	0.180046													
e(x)	-50	10000	1.00E-05	1	10050	0	8	10050	0.001008	8	10050	0.000999	8	10050	0.001001	0													
e(x)	-100	0	1.00E-05	1	-0.242495	9.5801	8	-0.242495	2.90782	8	-0.242495	12.8111	8	-0.242495	5.24277	2.90782													
e(x)	-100	100	1.00E-05	1	99.7575	9.54912	8	99.5177	2.98788	8	99.5177	12.0045	8	99.5177	5.23952	2.98788													
f(x)	-100	100	1.00E-03	1	4996.48	5.71744	8	4996.48	2.5944	8	4996.36	6.70407	8	4996.48	3.26999	2.5944													
f(x)	-100	100	1.00E-04	1	5001.9	60.64	8	5001.78	25.762	8	5001.78	64.8129	8	5001.78	32.3913	25.762													
f(x)	-100	0	1.00E-04	1	1.89899	58.127	8	1.89899	14.5857	8	1.89899	70.07	8	1.89899	26.6646	14.5857													
g(x)	-30	1000	1.00E-03	1	1.25E+09	1.28676	8	1.25E+09	1.1491	8	1.25E+09	1.35567	8	1.25E+09	1.22124	1.1491													
g(x)	-30	1000	1.00E-04	1	1.25E+09	10.092	8	1.25E+09	9.31345	8	1.25E+09	11.1183	8	1.25E+09	15.0607	9.31345													
g(x)	-30	0	1.00E-04	1	1.73023	9.96259	8	1.73023	4.76196	8	1.73023	11.2569	8	1.73023	5.44182	4.76196													
Tempo Total:					169.739124	Tempo Total:					65.259649	Tempo Total:					196.429159	Tempo Total:					97.011605	Tempo Total:					65.248857
Aceleração:					100.00%	Aceleração:					260.10%	Aceleração:					86.41%	Aceleração:					174.97%	Aceleração:					260.14%

Figura 5. Casos de teste e resultados

Em tarefas mais curtas, como funções lineares ou pequenos intervalos, podemos ver que o overhead que a computação concorrente traz não vale a pena em termos de desempenho, onde seu tempo foi tão pequeno que foi arredondado para zero, enquanto os métodos concorrentes tiveram um tempo pequeno, mais ainda sim maior que o sequencial.

Agora para tarefas maiores podemos observar que o overhead da computação concorrente vale muito a pena para a maioria dos programas que realmente exigem da nossa CPU.

Infelizmente, o balanceado normal ficou bem atrás do sequencial, acreditamos que isso seja devido a quase sempre o mutex do buffer estar ocupado e por isso não houve ganho nenhum, muito pelo contrário, demorou mais que o sequencial.

Com nossa pequena modificação dos múltiplos buffers confirmamos realmente que era o acesso ao buffer que estava atrasando o programa, conseguindo um desempenho substancialmente melhor que o de buffer único e o sequencial.

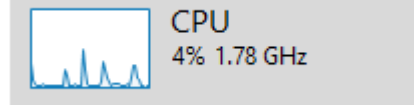
Para a nossa surpresa, o desbalanceado conseguiu o melhor ganho de desempenho, com uma aceleração de 260% de acordo com a Lei de Ahmdal, acreditamos que seja devido ao seu simples código e dispensa de estruturas adicionais, superando os balanceados até em casos que não deveria funcionar (como no segundo caso de teste da função g(x)).

Acreditamos que o de buffer múltiplo ainda possa ser melhorado com uma estrutura de dados diferente, pois uma pilha tem os retângulos próximos da precisão no final, e ao roubar retângulos do final, como nossa função **RoubaRetangulos** faz, as vezes nem estamos obtendo um ganho de desempenho significativo.

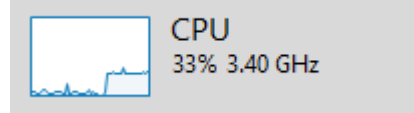
Garatindo o balanceamento:

Através de um monitor de recursos (TOP para linux ou Gerenciador de Tarefas para windows) podemos observar a utilização da CPU, observe:

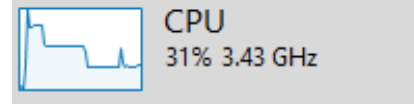
Inativo: (Sem nenhum programa rodando)



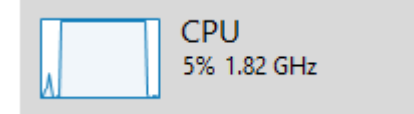
Sequencial: Perceba que realmente não há nada demais, o sistema operacional decide em qual thread ele irá rodar e ele permanece nela até o fim:



Desbalanceado: (Note o uso do processador caindo a medida que threads vão terminando)



Balanceado: Note que o processador está sempre totalmente ocupado, porem de forma não tão eficiente, visto que não produz tempos melhores



Balanceado c/ Múltiplos Buffers: Aqui podemos notar que o método não está de fato balanceado, visto que roubar um retângulo por vez deixa muitas threads inativas, talvez com uma outra estrutura de dados ou outra lógica de implementação poderíamos garantir um balanceamento e desempenho melhor

