Universidade Federal do Rio de Janeiro

Bacharelado em Ciência da Computação

Computação Concorrente



**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:

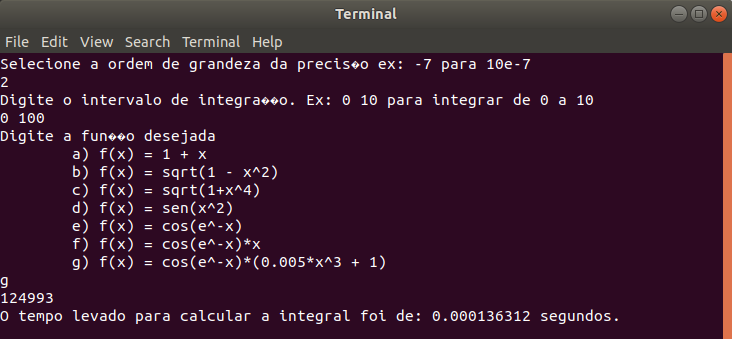
* Função que calcula o ponto médio
* Função que retorna o módulo
* Struct para representar os retângulos
* Função para o cálculo da integral
* Eventuais variáveis auxiliares como para definir a margem de erro

*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*



* **Versão concorrente desbalanceada:**

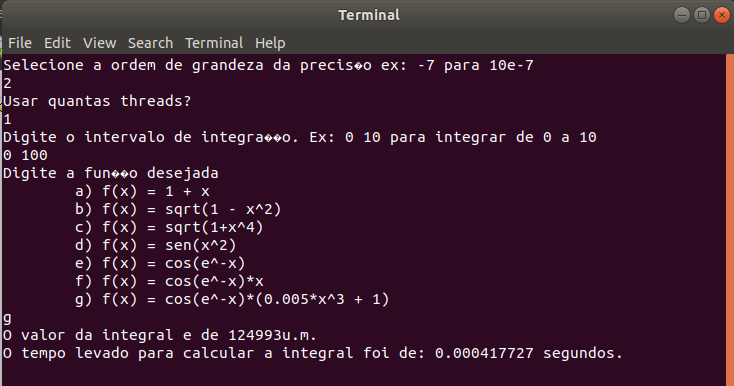
Nessa versão, foram adotadas as seguintes estratégias

* 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 hajam 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.*

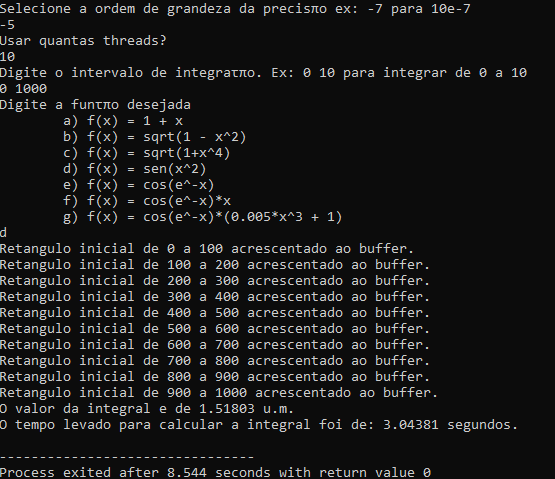


* **Versão concorrente balanceada**

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** (umpara **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 retangulos 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 retangulo preciso, ela adiciona esse valor a sua soma, caso não, ela empilha mais dois retangulos no buffer para serem processados, até a pilha ficar vazia sinalizando que a todos os retangulos foram consumidos e a integral está completa.*



* **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*

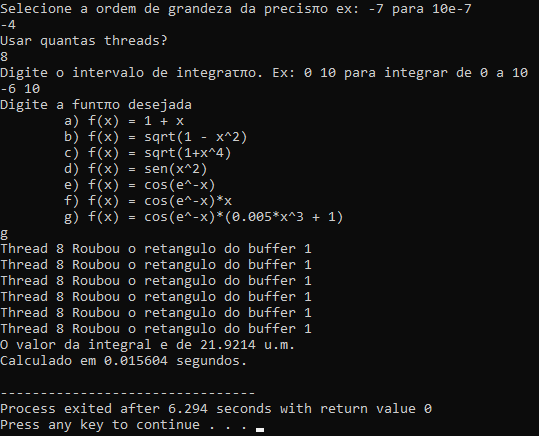
*O Array de Mutexes, que no caso tem tamanho oito, foi criado para reduzir a frequencia de acesso a um buffer individual significativamente, quase como uma tabela hash.*

*A função RoubaRetangulos serve para mantear 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.*

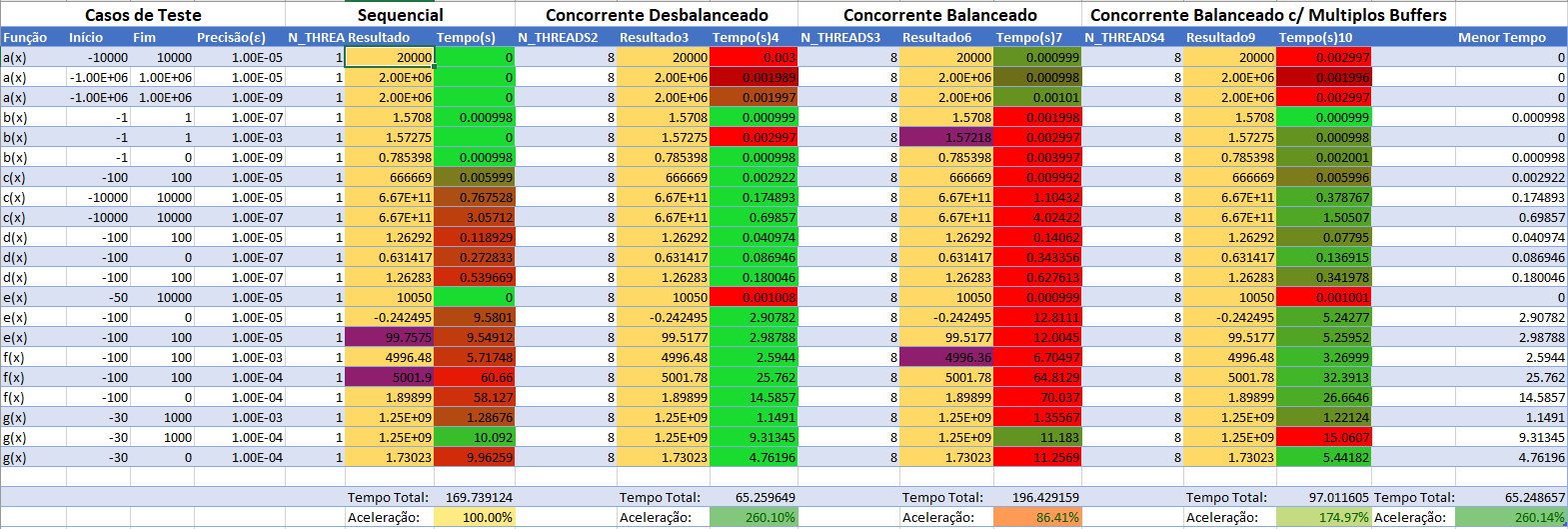
***Fluxo de execução do programa***

*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 númerica terminou.*



**Resultados:**

*Abaixo segue uma planilha dos casos de teste e seus resultados, ela pode ser conferida e modificada através do arquivo* ***Resultados.xlsx.***

**

*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.*

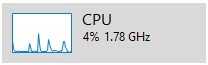
*Com nossa pequena modificação dos múltiplos buffers confirmamos realmente que era o acesso ao buffer que estava atrasando o programa, conseguindo um desempenho substancialmente melhor que o de buffer único e o sequencial.*

*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) ).*

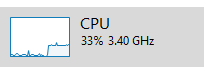
*Acreditamos que o de buffer múltiplo ainda possa ser melhorado com uma estrutura de dados diferente, pois uma pilha tem os retangulos proximos da precisão no final, e ao roubar retangulos do final, como nossa função* ***RoubaRetangulos*** *faz, as vezes nem estamos obtendo um ganho de desempenho significativo.*

**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 ):*

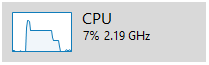


*Sequencial : Perceba que realmente não há nada demais, o sistema operacional decide em qual thread ele irá rodar e ele permanece nela até o fim*



*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 retangulo por vez deixa muitas threads inativas, talvez com uma outra estrutura de dados ou outra lógica de implementação poderiamos garantir um balanceamento e desempenho melhor*