Primeiro Trabalho - Relatório Final

Universidade Federal do Rio de Janeiro

Disciplina: Computação Concorrente (MAB-117)

Período: 2021/1 Remoto

Título do Trabalho: Tabuadas de Multiplicação i x j

Nomes dos Membros: Gabriel Almeida Mendes, Luiz Rodrigo Lacé Rodrigues

DRE dos Membros: 117204959, 118049783

1. Projeto final e implementação da solução concorrente

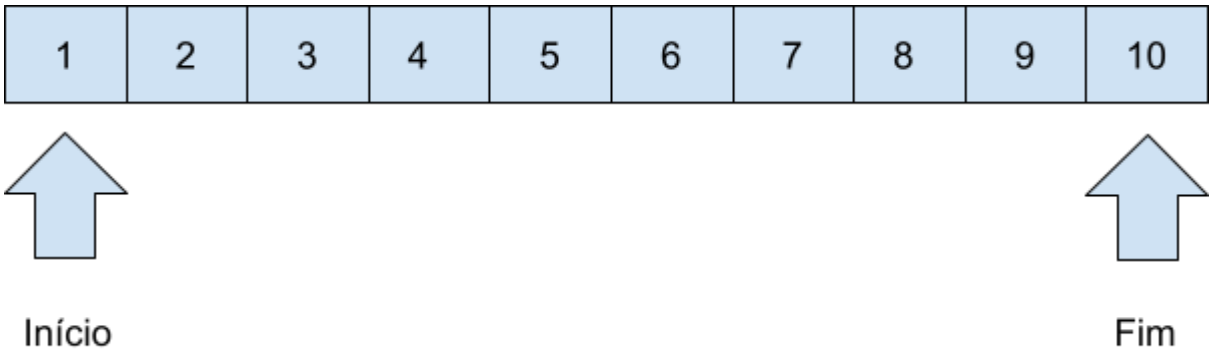
O programa tem o objetivo de criar e imprimir uma matriz que é uma tabela de multiplicação por coordenadas cartesianas. Essa tabela ilustra cálculos de várias tabuadas de Pitágoras que vai de um até um número de tabuadas digitado pelo usuário. Como exemplificado no Relatório Parcial.

A implementação com solução concorrente surgiu da necessidade de dividir o cálculo das tabuadas entre as threads de forma que haja ganho de desempenho de tempo, focado principalmente para quantidade de números maiores que a ordem de n^6.

Criamos uma função thread que divide a quantidade de tabuadas entre as threads, às calcula. Inicialmente pensamos em retornar um vetor com todas contas em ordem e na função main, todos esses vetores gerados pelas threads separadamente seriam colocados num vetor global ainda na função pthread_join. Entretanto optamos em colocar os resultados diretamente em um vetor global por motivos que serão melhor detalhados na parte de discussão.

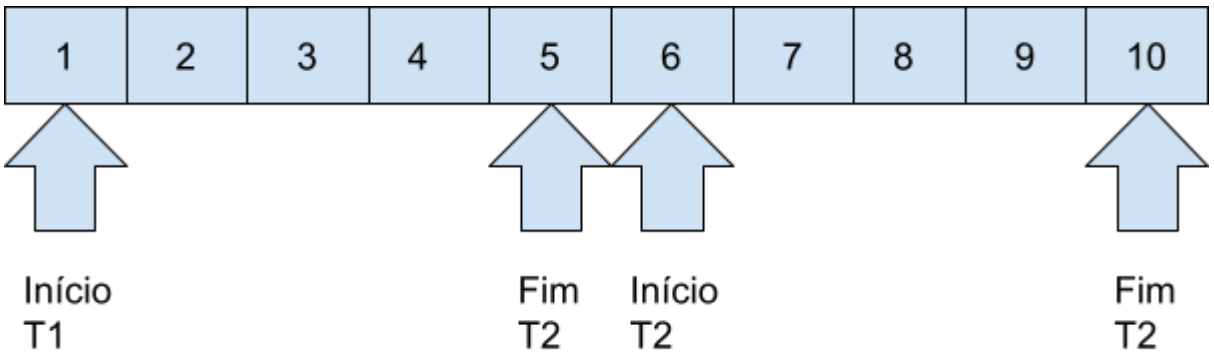
A ideia aqui é de adiantar os cálculos das tabuadas em blocos que serão baseados nos números de tabuadas e threads passados como parâmetros

Em uma implementação sequencial os cálculos podem ser ilustrados dessa forma:



No desenho acima podemos ignorar os múltiplos (x1, x2, x3, etc) da tabuada e vamos ter em mente apenas as tabuadas (1x, 2x, 3x, etc),visto que nas threads vamos calcular todos os múltiplos de cada tabuada que vai ser tratada naquela thread. Na forma sequencial precisamos calcular as tabuadas de 1 a 10 (independente do múltiplo de forma que a última tabuada só vai ser calculada se as suas anteriores também forem.

Entretanto quando partimos para o caso concorrente do qual implementamos, se dividirmos essa tarefa em 2 threads, vamos chegar no seguinte caso



Aqui a última tabuada não vai mais precisar esperar tanto para ser calculada, visto que seu cálculo não depende mais das 9 anteriores.

Para verificarmos a corretude do programa, simplesmente iteramos dois vetores, um que foi implementado de forma sequencial e outro de forma concorrente e comparamos o valor na mesma posição, se os valores fossem diferentes, retornaríamos um erro.

No fim, os valores do vetor global serão passados para uma matriz de coordenadas cartesianas, que utilizando 10 tabuadas com 10 múltiplos cada, chegaríamos em algo parecido com isso

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

2. Casos de teste

Devido a baixa complexidade do nosso projeto, alguns casos de teste tiveram que ser alterados visto que para casos com número pequenos a versão sequencial ainda se demonstrava com um tempo de execução menor que em relação a versão concorrente. Isso pode ser verificado nos casos 1, 2 e 3.

Mas vamos alterando os casos para vermos como é possível chegar em um ganho com o concorrente.

Para os seguintes tempos, fizemos 5 execuções e pegamos o menor tempo entre elas.

A diferença dos casos de concorrentes 1 thread para o sequencial é que o concorrente passa por processos diferentes para chegar no mesmo resultado, o que causa um gasto diferenciado de memória e processamento

Caso 1 - nTabuadas = 100; nMultiplos = 100; nThreads = 1

Avaliação de Corretude: Confirmado
Desempenho Sequencial: 0.000054
Desempenho Concorrente: 0.000250

Caso 2 - nTabuadas = 100; nMultiplos = 100; nThreads = 2

Avaliação de Corretude: Confirmado
Desempenho Sequencial: 0.000064s
Desempenho Concorrente: 0.000196s

OBS: Para o caso 3, podemos ver um claro exemplo de que aumentar o número de threads para tratar poucos casos pode não ser vantajoso, pelo contrário. Estamos dividindo poucas tarefas para mais threads que o necessário, o que começa a gerar um aumento no tempo concorrente.

Caso 3 - nTabuadas = 100; nMultiplos = 100; nThreads = 4

Avaliação de Corretude: Confirmado
Desempenho Sequencial: 0.000064s
Desempenho Concorrente: 0.000318s

OBS: Agora nos casos 4, 5 e 6 começamos a ver uma aproximação dos números, visto que agora a implementação concorrente começa a fazer mais sentido, visto que estamos trabalhando com uma quantidade maior de números.

Caso 4 - nTabuadas = 500; nMultiplos = 300; nThreadas = 1

Avaliação de Corretude: Confirmado
Desempenho Sequencial: 0.000911
Desempenho Concorrente: 0.001687

Caso 5 - nTabuadas = 500; nMultiplos = 300; nThreadas = 2

Avaliação de Corretude: Confirmado
Desempenho Sequencial: 0.000795
Desempenho Concorrente: 0.001249

OBS: No caso 6 chegamos em um estado onde a solução concorrente se demonstra mais vantajosa que a sequencial.

Caso 6 - nTabuadas = 500; nMultiplos = 300; nThreadas = 4

Avaliação de Corretude: Confirmado
Desempenho Sequencial: 0.000912
Desempenho Concorrente: 0.000782

Caso 7 - nTabuadas = 1000; nMultiplos = 1000; nThreads = 1

Avaliação de Corretude: Confirmado
Desempenho Sequencial: 0.005641
Desempenho Concorrente: 0.006106

Caso 8 - nTabuadas = 1000; nMultiplos = 1000; nThreads = 2

Avaliação de Corretude: Confirmado
Desempenho Sequencial: 0.007319
Desempenho Concorrente: 0.004800

Caso 9 - nTabuadas = 1000; nMultiplos = 1000; nThreads = 4

Avaliação de Corretude: Confirmado
Desempenho Sequencial: 0.005663
Desempenho Concorrente: 0.003532

OBS: Para valores muito acima deste último, o programa começa a travar ou não funcionar, sempre alegando falha de segmentação.

Em resumo temos uma tabela do que obtivemos:

	nthreads: 1	nthreads: 2	nthreads: 4
100x100	Sequencial: 0.000054 Concorrente: 0.000250	Desempenho Sequencial: 0.000064s Desempenho Concorrente: 0.000196s	Desempenho Sequencial: 0.000064 Desempenho Concorrente: 0.000318
500x300	Desempenho Sequencial: 0.000911 Desempenho Concorrente: 0.001687	Desempenho Sequencial: 0.000795 Desempenho Concorrente: 0.001249	Desempenho Sequencial: 0.000912 Desempenho Concorrente: 0.000782
1000x1000	Desempenho Sequencial: 0.005641	Desempenho Sequencial: 0.007319	Desempenho Sequencial: 0.005663

	Desempenho Concorrente: 0.006106	Desempenho Concorrente: 0.005800	Desempenho Concorrente: 0.003532

3. Avaliação de desempenho

Aqui está a configuração da máquina onde os testes foram realizados;

Sistema Operacional: "20.04.3 LTS (Focal Fossa)", Ubuntu, Linux.

Arquitetura: x86_64
Modo(s) operacional da CPU: 32-bit, 64-bit
Ordem dos bytes: Little Endian
Address sizes: 43 bits physical, 48 bits virtual
CPU(s): 8
Lista de CPU(s) on-line: 0-7
Thread(s) per núcleo: 2
Núcleo(s) por soquete: 4
Soquete(s): 1
Nó(s) de NUMA: 1
ID de fornecedor: AuthenticAMD
Família da CPU: 23
Modelo: 24
Nome do modelo: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx

Como as medidas de tempo estavam muito próximas na maioria dos casos, preferimos adotar o método de executar cada casa pelo menos 5 vezes e anotar o menor tempo entre eles. Para facilitar a análise dos dados utilizamos a macro “timer.h” e tomamos o tempo de execução do programa em segundos, e analisamos suas casas decimais.

Utilizamos a lei de Amdahl para estimar os ganhos de desempenho, onde o ganho de velocidade é calculado por Tsequencial/ts + tp(P). Temos:

ts: tempo da parte sequencial do programa (que não será dividida entre threads)
tp(P): tempo da parte paralela do programa usando P processadores
T(P): tempo total de execução previsto do programa usando P processadores (ts + tp(P))

Como estamos usando apenas um processador temos os seguintes cálculos:

Cálculos da aceleração;
100x100, 2 threads: 0.000054/0.000196 = 0,275510204
100x100, 4 threads: 0.000054/0.000318 = 0,169811321

500x300,2 threads: 0.000795/0.001249 = 0,636509207
500x300,4 threads: 0.000795/0.000782 = 1,016624041

1000x1000, 2 threads: 0.005641/0.004800 = 1,175208333
1000x1000, 4 threads: 0.005641/0.003532 = 1,597112118

Ganhos em relação a versão sequencial:

	nthreads: 2	nthreads: 4
100x100	0,275510204	0,169811321
500x300	0,636509207	1,016624041
1000x1000	1,175208333	1,597112118

Os números em vermelho representam os casos que é desvantajoso utilizar a implementação concorrente, enquanto os de verde mostram o ganho de desempenho.

4. Discussão

No início do desenvolvimento deste trabalho nosso raciocínio quanto ao ganho de desempenho mudou duas vezes. Na primeira implementação, quando utilizamos uma struct para retornar das threads vetores com os resultados das tabuadas e também a quantidade de valores para que fosse possível fazer a iteração em outro vetor na main o nosso programa não estava dando ganho algum em relação a versão sequencial.

Em uma breve comunicação com a professora, descobrimos que a nossa implementação utilizando struct e passando de um vetor de resultados parciais para um vetor “global” estava consumindo um tempo desnecessário, visto que o que a implementação sequencial já colocava direto todos os resultados no vetor sem precisar passar de um vetor para outro.

Passamos então para uma implementação onde as threads não retornavam mais uma struct com a quantidade de valores e um vetor com resultados parciais, agora elas passam os resultados direto para um vetor global. Como as posições dos vetores possuem endereços diferentes, não precisamos utilizar de métodos da exclusão mútua quando as threads forem acessá-los.

Quando passamos para os casos de teste da nossa versão atual do programa, começamos por valores baixos, o resultado nos pegou de surpresa visto que a versão sequencial ainda era mais rápida que a concorrente. Como não éramos o que esperávamos, passamos a pesquisar qual era o motivo disso pois estávamos dividindo as tarefas, como seria possível que a mesmo assim ainda fosse mais lento? A resposta estava na complexidade do nosso problema, os computadores atuais conseguem fazer multiplicações praticamente de forma instantânea e calcular uma quantidade pequena de tabuada de forma concorrente não teria tanta vantagem em relação a sequencial visto que não precisaria passar por todos os custos de criar threads, passar pelo algoritmo delas e ainda as receber na main.

Percebemos também o caso onde, além das poucos valores a serem calculados, quando dividimos a tarefa em uma quantidade maior de threads, o processo ficava cada vez mais lento, o que demonstra que a programação multithreading não gera um ganho linear em relação a quantidade de threads e sim possui o seu valor "ótimo" para o melhor ganho possível.

Entretanto, apesar da frustração, começamos a analisar os casos para números cada vez maiores e a surpresa foi positiva. O ganho de desempenho começou a se demonstrar para a quantidade de valores da ordem de n^4, mostrando que no final a nossa ideia inicial de que poderiam haver ganhos estava sim correta, só não tínhamos ideia que não seriam para todos os casos.

Ao chegarmos nos valores da ordem de n^6 o programa começa a dar ou erro de malloc ou de segmentação, o que nos impossibilitou de analisar mais casos com um “gap” maior. Então com a disponibilidade de um hardware melhor seria possível demonstrar os ganhos de desempenho de forma mais clara. Quanto ao refinamento do código, acreditamos que algum raciocínio quanto a printar a matriz final por meio das tarefas das threads poderia vir a nos dar alguma vantagem, entretanto precisaríamos utilizar uma forma sequencial para isso e por enquanto não há forma clara de fazer isso de forma diferente.

As dificuldades no trabalho, que apesar de simples, foram inúmeras pois entramos em um campo de problemas abstratos que não eram tão evidentes de entender apenas olhando o código. Entender como dividir as threads nos custou um certo tempo para desenvolver o raciocínio, mas a maior dificuldade que encontramos foi em entender porquê casos com valores pequenos podem não gerar nenhum ganho de desempenho, como foi falado anteriormente.

Por fim, acreditamos que o trabalho nos gerou mais conhecimento do que imaginamos devido às questões abstratas e técnicas quanto ao funcionamento do computador. Acreditamos também que é um ótimo exemplo que pode ser utilizado de forma didática que iniciantes da computação e da programação poderiam entender sem grandes dificuldades.

5. Referências bibliográficas

Aulas gravadas da disciplina de Computação Concorrente.

Discussão quanto ao ganho das threads:

https://stackoverflow.com/questions/28436001/c-pthreads-multithreading-slower-than-single-threading

6. Repositório para o código da implementação:

https://gitlab.com/luizrodrigolace/laboratiorios-computacao-concorrente/-/blob/master/trabalhos/trabalho%201/tabconc.c