24 de agosto de 2021

Universidade Federal do Rio de Janeiro

Sistemas Distribuidos

Trabalho 2

Bruno Dantas de Paiva
Rodrigo Mendes Palmeira

1 Escolhas de Implementação

1.1 Escolha da Linguagem de Programação

Decidimos utilizar a linguagem Go para o projeto por termos um interesse prévio de aprender a linguagem somado ao seu bom balanço de capacidade de realizar ações de baixo nível, necessárias para o projeto, e certas facilidades modernas como garbage collection. Ademais, Go foi feito pensando em sua aplicação em sistemas distribuídos com um tipo de primitiva especial chamada de *Go routines*(Rotinas GO) que possibilitam o uso de CSP(Communicating Sequential Processes)[1] que achamos que pode ser bastante interessante de utilizar para os trabalhos da disciplina. Além disso o código está disponível no github pelo link `https://github.com/DantasB/Distributed-Systems/tree/main/Trabalho_2`

2 Implementações

2.1 Spinlocks

2.1.1 syncprim

Para a implementação do spinlock, utilizamos um pacote da biblioteca padrão do go chamado atomic que provê uma série de funções atômicas e é usado para construir primitivas de sincronização definidas na biblioteca padrão. Assim, fizemos um pacote simples, inspirado nos slides da disciplina, chamado syncprim com duas função *Aquire* e *Release* que manipulam uma variável chamada *lock*, a *Aquire* utiliza-se da função atômica *CompareAndSwapUint32* para funcionar corretamente.

2.1.2 Somador

Para a geração do números aleatórios, fizemos uma função que utiliza uma função da biblioteca padrão de go que gera um número entre 0 e 100 multiplicando esse por 1 ou -1 a depender do resultado da chamada de outra função aleatória. O programa recebe como argumento de linha de comando o tamanho do vetor e o número de threads, inicializa-o com os valores aleatórios gerados a partir da função supracitada, divide-o em $k - 1$ partições

[1] https://www.cs.cmu.edu/ crary/819-f09/Hoare78.pdf

iguais e uma k-éssima para o resto(da divisão n/k) e cria uma thread para executar a função de soma em cada uma dessas partições, rodamos 10 vezes essa última parte medindo o tempo decorrido e tirando a média. No final o programa exibe os valores de n e k e o tempo médio decorrido durante a execução da parte concorrente do programa.

2.1.3 Estudos de Caso

Rodamos os estudos de caso para diferentes valores de N e k conforme especificado no trabalho utilizando um script bash e geramos o seguinte gráfico:

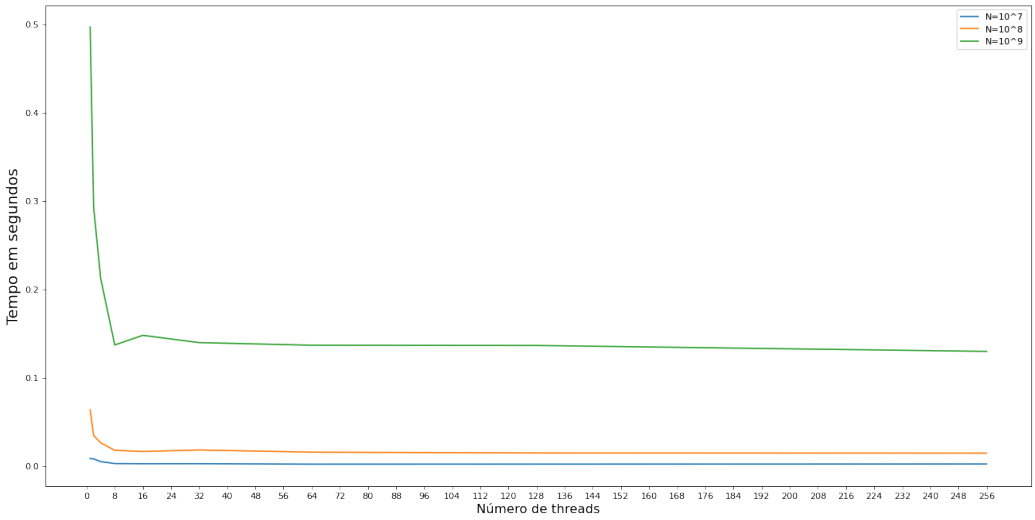

Figura 1: Execução da parte concorrente do somador

Podemos perceber que, como esperado, o tempo de execução é sempre maior para N maiores. Mais interessante, é analisar a mudança do tempo de execução conforme o número de threads aumenta e podemos perceber que ele diminui de forma significativa até 8 threads e depois estabiliza para todo N. Isso faz sentido, pois o programa foi rodado em um computador cujo processador possui 8 cores, ou seja, até esse número há ganho de paralelismo com o aumento do número de threads. Poderia se esperar que o tempo de execução aumentasse com o número de threads a partir de 8 pelo overhead introduzido pela criação delas, porém isso não acontece, pois as threads em go, chamadas mais comumente do *go routines*, são user threads que são mapeadas para um número de kernel level threads definido pelo runtime

da linguagem, a partir do número de cores do processador onde o programa está rodando.

2.2 Produtor Consumidor

Para produtor consumidor, foi utilizado como base o código demonstrado em aula de produtor e consumidor, a fim de otimizar e evitar condições de corridas que iriam afetar o resultado obtido. Além disso, como boas práticas de programação, foram criadas inúmeras funções auxiliares (que não foram reutilizadas) a fim de facilitar a leitura e entendimento do código, como abordaremos melhor nos tópicos abaixo.

2.2.1 Produtor

Para o código de produtor não houve muita dificuldade, era necessário somente verificar se existia algum espaço vazio na memória, a fim de alocar este ponto de memória com um número aleatório.

Um ponto interessante sobre a linguagem escolhida e que os semáforos necessitam de ser criados com um tamanho específico, no qual optamos por utilizar o próprio tamanho do input N (tamanho da memória) para facilitar a implementação.

Além disso, os semáforos em golang necessitam de um contexto para serem criadas. Contextos são interfaces que permitem que o usuário envie signals à outras goroutines (user-level threads), porém estas não foram utilizadas externamente, somente para a instanciação do próprio semáforo.

2.2.2 Consumidor

Para o código de consumidor foi necessário modificar um pouco o código encontrado em aula pois um dos requisitos era parar todas as threads ao alcançar um ponto determinado, em nosso caso quando o número de valores consumidos alcançasse 10^5. Tendo isto em mente, foi necessário adicionar um outro semáforo (loopControl) que permitiu que nós checássemos o valor da variável de controle de loop M (número máximo de números a serem consumidos), a fim de parar todas as threads quando esse valor alcançasse 0. Além disso, para "avisar"o término dessa condição, foi utilizado o channel, um objeto de comunicação entre goroutines, que permite que outra variável adquira esse valor de controle. Sabendo disso, criamos um channel finished

que ao receber exatamente nc mensagens true, indica que o código terminou e encerra a execução.

2.2.3 Execução

Para a execução do código do produtor consumidor, foi criado um script auxiliar em bash, com o objetivo de automatizar a variação dos inputs e também estruturar todos os resultados em um arquivo .csv que foi utilizado para gerar um gráfico por meio de um programa auxiliar em python. Ademais, é importante ressaltar que um dos requisitos deste trabalho era a impressão da mensagem, por parte do consumidor a impressão se o valor obtido da memória é primo ou não. Pelo fato de estarmos executando em batches, todos os resultados impressos pelo código são armazenados em arquivos separados cada execução do código para cada input do usuário.

2.2.4 Estudos de Caso

Pensando que o sistema operacional executa vários processos secundários, e que estes processos, em algumas linguagens, podem vir a atrapalhar a execução do código justamente por conta do escalonamento fazendo com que esses códigos tendam a demorar mais e influenciar no resultado, foram escolhidas duas formas de execução.

Antes de mostrar as execuções de fato, é necessário ter em mente que para facilitar a criação de uma visualização, foi utilizada uma parametrização das tuplas do número de threads de produtor e consumidor. A relação pode ser encontrada na tabela abaixo:

Índice	**(Número de threads Produtor, Número de threads Consumidor)**
0	(1,1)
1	(1,2)
2	(1,4)
3	(1,8)
4	(1,16)
5	(2,1)
6	(4,1)
7	(8,1)
8	(16,1)

Figura 2: Parametrização Índice x número de threads

A primeira foi a execução de um usuário comum, ou seja, executando em código enquanto fazia outras atividades (navegador aberto, dispositivos de comunicação aberto e etc), a fim de ter noção real do desempenho com influência externa, e foi obtido o seguinte gráfico:

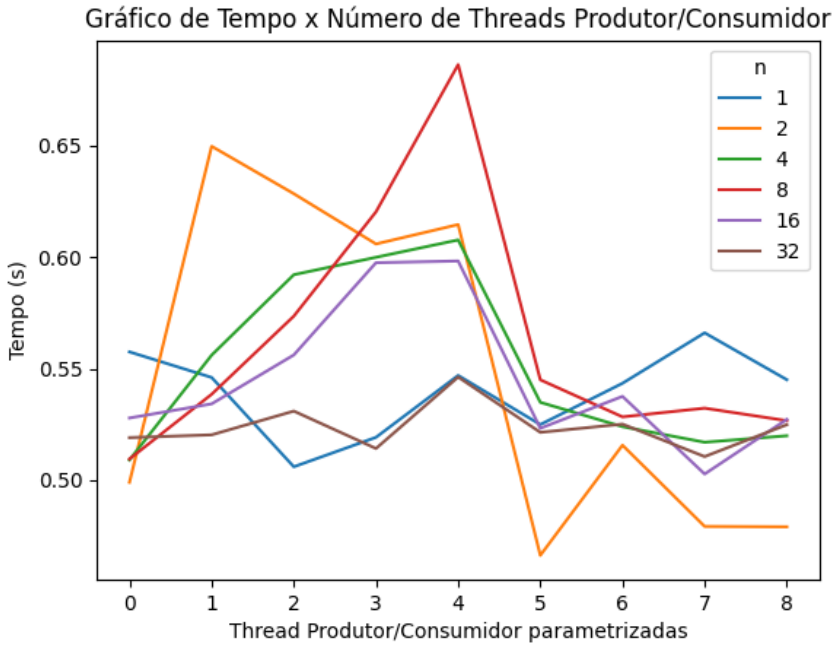

Figura 3: Execução do Produtor e Consumidor 1

A segunda foi a execução de um modo alternativo. O usuário, em alguns sistemas operacionais linux, podem acessar um terminal sem fazer o login, deste modo, um número menor de processos é executado, optamos por fazer isso para obter uma execução mais limpa sem influência de nenhum processo externo, sendo obtido o seguinte gráfico:

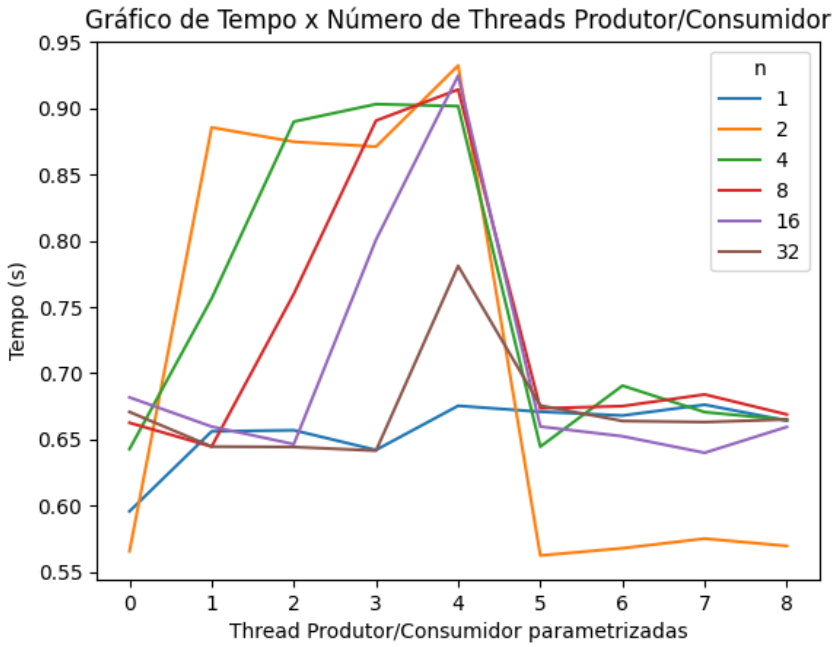

Figura 4: Execução do Produtor e Consumidor 2

Após analisar os dois casos, foram percebidos casos semelhantes. O primeiro ponto observado é que aumentar o número de threads consumidoras tende a melhorar o tempo de execução para grande parte dos casos, contudo isto não ocorre para 1 thread consumidora e 1 thread produtora. O segundo ponto é que quando o tamanho da memória é de valor 2, para a linguagem utilizada, possui-se um desempenho melhor comparado aos outros tamanhos de memória. Tal fato pode ser por conta das threads em golang serem user-level e a forma que a linguagem acaba gerenciando essas trocas de contexto, são favorecidas quando há um número específico de troca de contexto. O terceiro e último ponto foi que, em golang, a linguagem atua melhor quando possui outros processos rodando paralelamente. Para este ponto, pesquisamos um pouco mais a fundo a respeito de como a linguagem atua porém não temos explicações claras a respeito do motivo disto acontecer.