**UFRJ**

**UNIVERSIDADE FEDERAL DO RIO DE JANEIRO**

Departamento De Ciência Da Computação

Disciplina: Computação Concorrente

# Relatório Sobre Leitura E Processamento De Arquivos Usando Threads

Alunos:

Abraham Banafo Ampah - 117074396

Leonardo Veiga – 117048719

28 de fevereiro de 2021

SUMÁRIO

# INTRODUÇÃO

Para este trabalho, utilizamos 4 threads além da thread main, onde uma thread é utilizada somente para leitura de um arquivo binário e 3 threads são utilizadas para encontrar os padrões, enquanto a main espera pelo término das 4 threads e imprime o resultado. A lógica utilizada nas 3 threads que buscam os padrões é basicamente a mesma, mudando somente a parte onde se computa os padrões. Foi utilizada também a sincronização por semáforos onde tivemos um desafio para poder sincronizar as threads de forma eficiente. Encontramos um jeito, com semáforos, de sincronizar, porém não é tão eficiente quanto o uso de *locks*. A nossa ideia para construir o programa foi utilizar o problema leitor / escritor com algumas pequenas modificações.

# DESENVOLVIMENTO

### Objetivo:

Construir um programa concorrente onde lemos o conteúdo de um arquivo binário para buscar padrões pré-especificados.

### Justificativa:

Nesta seção, iremos falar mais sobre o código e as escolhas feitas para a implementação. Primeiro, discutiremos a thread que lê e escreve no buffer e depois passaremos para as outras threads.

Inicialmente, a thread de leitura irá consumir os N primeiros bytes do arquivo binário e procurará pelo primeiro número da sequência que indicará quantos números precisam ser lidos. Não tratamos o caso em que esse número esteja dividido em dois ou mais blocos, ou seja, o parâmetro N deve poder conter esse primeiro número na primeira leitura. Caso contrário, ele só converterá para inteiro a parte mais significativa. Como estamos tomando um N relativamente grande, então o arquivo de entrada precisaria ter muitos espaços em branco antes do primeiro número da sequência para que haja algum problema no código. Depois que descobre quantos números devem ser lidos, limpamos esse número no buffer e atualizamos as variáveis de controle (quantos blocos faltam ser lidos, se ainda há números para escrever no buffer, etc). Daí, entramos em um ciclo que acaba sob duas condições: chegamos no final do arquivo ou lemos todos os números.

As threads que leem o buffer a fim de encontrar os padrões possuem duas partes comuns: as funções **testaTermino()**, no início, e **atualizaBlocos(int)**, no fim. A primeira é somente para terminar a execução das threads, enquanto que a segunda atualiza a quantidade de blocos lidos em comum pelas threads, liberando o escritor para escrever mais blocos. Caso haja pelo menos uma thread que não leu ainda, então a função não faz nada, e caso a thread que chamou a função leu todos os blocos disponíveis, então ela deve aguardar outras threads atualizarem os blocos para que ela possa ser liberada. O código entre a chamada dessas funções tem o objetivo de computar os padrões de cada thread.

Para a sincronização, a melhor forma que encontramos foi desbloqueando uma thread de cada vez, seguindo a ideia do laboratório para semáforos. Assim, as threads leitoras desbloqueiam o escritor e elas mesmas, enquanto que o escritor desbloqueia somente uma thread leitora. De fato, tivemos algumas chamadas a mais de funções para

sincronização do que teríamos usando *locks*, além de termos que desbloquear uma thread por vez, já que a nossa tentativa de simular um *broadcast* com semáforos não deu certo.

**Funções Criadas:**

**void* leArquivo(void* arg):**

Thread lê arquivo e escreve no buffer. A primeira escrita no buffer é tratada de forma que possamos saber qual a quantidade de números que são lidos.

**void* iguais(void* arg):**

Thread que procura pela maior sequência de números iguais no arquivo.

**void* triplas(void* arg):**

Thread que computa as sequências de 3 números iguais consecutivos.

**void* sequencia(void* arg):**

Thread que computa as sequências <012345>.

**long long int leitura(FILE*, long long int*):**

Função utilizada para acessar o arquivo e escrever no buffer.

**void testaTermino():**

Verifica se a thread **leArquivo** terminou de escrever e não há mais blocos para serem lidos.

**void atualizaBlocos(int):**

Função usada para atualizar a quantidade de blocos lidos. Utilizamos a thread mais lenta na leitura como parâmetro para descobrir quantos blocos foram lidos por todas as threads.

**void inicializaBuffer()**

Função auxiliar que armazena espaço para os blocos lidos.

**void liberaBuffer():**

Função auxiliar que libera o espaço utilizado pelo buffer.

## CONCLUSÃO (Testes e Análise)

Na medida do tempo, o cabeçalho timer.h foi usado para medir o tempo de execução. Este arquivo está disponível no repositório do Github.

Os experimentos foram feitos em um pc com 4 processadores (CPU AMD A8 9600)

Os testes foram repetidos 3 vezes para cada dimensão.

**Tabela 1: Tempo (em segundos) de execução com 4 threads, buffer igual 3(M = 3) e blocos igual 500 (N = 500)**

| Dimensão/Threads/Vezes | 4/1st | 4/2nd | 4/3rd | Média |
|---|---|---|---|---|
| 30 | 0.00212789 | 0.00304008 | 0.00231194 | 0.00249330 |

| | | | | |
|---|---|---|---|---|
| 60 | 0.00209498 | 0.00193906 | 0.00328398 | 0.00243934 |
| 300 | 0.00193810 | 0.00190187 | 0.00288415 | 0.00224137 |
| 3000 | 0.00230408 | 0.00229597 | 0.00219297 | 0.00226434 |
| 1000000 | 0.11238408 | 0.11352801 | 0.10644197 | 0.11078468 |

Baseado nos testes feitos, os resultados obtidos em cada um dos testes foram resultados corretos. Isso nos dá certeza da funcionalidade do programa. A execução com uma thread não foi possível pois estamos obrigados a usar threads diferentes para outras partes do programa.

Também percebemos que houve uma melhoria no tempo quando o tamanho de buffer foi incrementado de 3 para 20

**Tabela 2. Tempo com buffer igual a 20 ( M=20 ) e blocos igual 500 (n=500)**

| Dimensão | 1st | 2nd | 3rd | Média |
|---|---|---|---|---|
| 1000000 | 0.05708790 | 0.06712198 | 0.06300306 | 0.06240431 |

Quando o tamanho dos blocos é decrementado , não houve melhoria no tempo, o tempo aumentou.

**Tabela 3. Tempo com blocos igual a 30 ( N=30 ) e buffer igual 3 (M=3)**

| Dimensão | 1st | 2nd | 3rd | Média |
|---|---|---|---|---|
| 1000000 | 0.49046397 | 0.50063109 | 0.63906193 | 0.54338566 |

Quando o tamanho de blocos e o tamanho de buffer foram aumentados foi percebido que houve uma melhoria no tempo.

**Tabela 4. Tempo com buffer igual a 50 ( M=50 ) e tamanho de blocos igual 1000 (N=1000)**

| Dimensão | 1st | 2nd | 3rd | Média |
|---|---|---|---|---|
| 1000000 | 0.04933619 | 0.04552817 | 0.04765701 | 0.04750712 |

Foi escolhida a dimensão 1000000 para o resto dos testes por que essa dimensão deu o maior tempo nos primeiros testes.