Universidade Presbiteriana Mackenzie

Ciência da Computação - Computação Paralela

# Projeto Prático 1 - Computação Paralela

Francisco Losada Totaro - 10364673

Pedro Henrique L. Morerias - 10441998

# Relatório:

## Introdução:

Diante do problema apresentado onde, dado um gerador de logs, era necessário a criação de um programa que analisasse esses logs, e que retornasse a quantidade de erros ‘404’ e o tempo total dessa execução, tanto de forma sequencial, quanto de forma paralela.

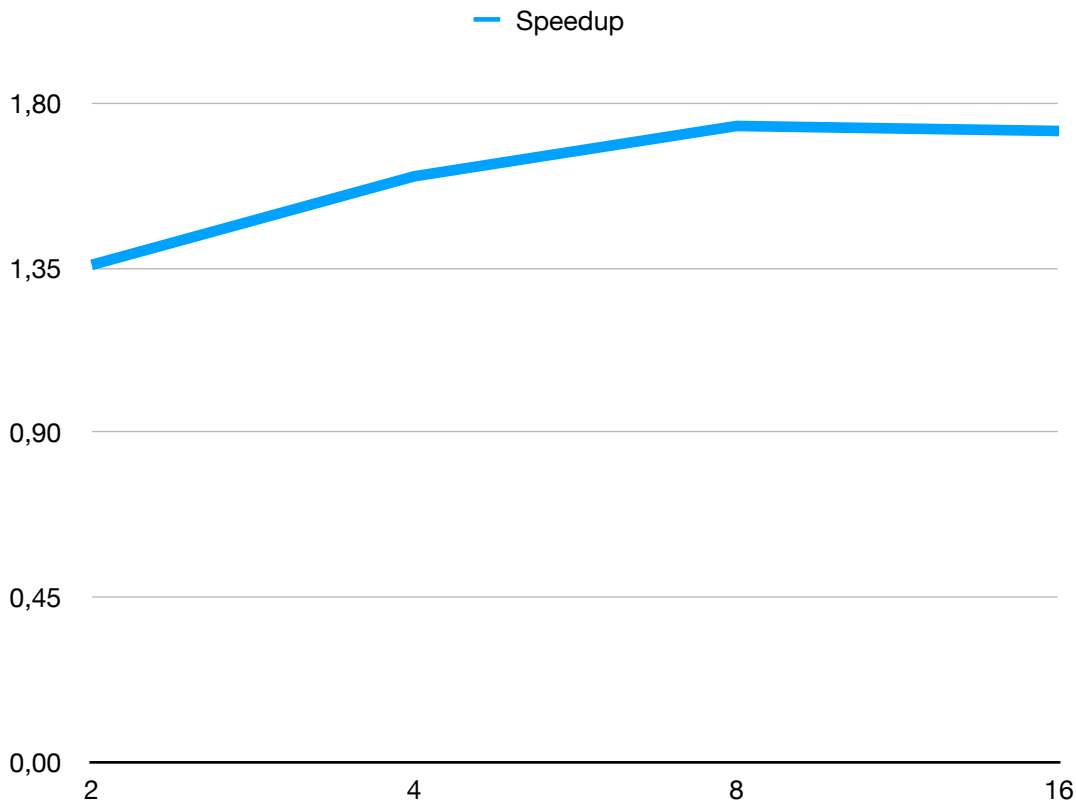
Para chegar a uma solução em paralelo, inicialmente, estruturamos o código de forma sequencial, assim como solicitado, além de obter um melhor entendimento de como construir uma solução base, que seria melhorada para o caso em paralelo. Para essa aplicação, nós realizamos um leitura no arquivo gerado e, enquanto a leitura das linhas ocorre, realizamos verificações onde, caso na posição da linha, onde está indicando o status, existir o valor ‘404’, a variável ‘count404’ será incrementada, para trazer o total de erros no final, e, caso o resultado for igual a ‘200’, a variável ‘count200Bytes' é incrementada, onde irá mostrar o número total de bytes transferidos em requisições bem sucedidas.

Sobre a estratégia paralela utilizada, assim como citado anteriormente, utilizamos a nossa solução sequencial como base. Para essa estratégia, inicialmente, é criada uma Struct ‘Stats’, onde irá armazenar os devidos valores totais de 404 e bytes de 200. Criamos as threads com o pthread_create e aguardamos todas entrarem com o pthread_join.

Agora, na função que as threads irão trabalhar, cada uma irá selecionar qual parte do arquivo cada ela irá trabalhar/ler, sempre sendo garantido que nenhuma outra thread está trabalhando nela. Garantimos isso quando fazemos o parse do arquivo com o strtok_r, que salva em que ponto a string terminou de ser lida. Após isso, são criadas variáveis locais, ‘local404' e ‘localBytes’, que irão armazenar os valores lidos localmente por cada uma das threads. Após a thread finalizar sua parte, é realizada um ‘phtread_mutex_lock’, onde serão passados os valores salvos nas variáveis locais para as globais, presentes na struct ‘Status’. Depois ocorre um ‘mutex_lock’, liberando as variáveis para outras threads utilizarem. No final do programa, quando todas as threads foram encerradas, apresentamos o resultado final e destruímos os Mutex.

# Resultados:

| Número de Threads | | | | | | Tempo de execução médio | Speedup | Eficiência |
|---|---|---|---|---|---|---|---|---|
| **1** | 0,205228 | 0,204368 | 0,203014 | 0,202270 | 0,20315 | 0,203606 | 1 | 1 |
| **2** | 0,151656 | 0,149242 | 0,149514 | 0,149793 | 0,149694 | 0,1499798 | 1,36 | 0,68 |
| **4** | 0,128402 | 0,128425 | 0,128035 | 0,126668 | 0,124878 | 0,1272816 | 1,60 | 0,40 |
| **8** | 0,115045 | 0,118051 | 0,120245 | 0,113842 | 0,118790 | 0,1171946 | 1,74 | 0,22 |
| **16** | 0,118182 | 0,117884 | 0,118351 | 0,118559 | 0,117670 | 0,1181292 | 1,72 | 0,11 |




# Conclusão:

A partir dos resultados obtidos do gráfico, é possível observar que, conforme aumentamos o numero de threads, o Speedup aumenta, porém esse aumento vai diminuindo conforme a quantidade de threads. Como aumentamos o número de threads, sem aumentar o tamanho do problema, e a eficiência diminuiu, então podemos dizer que ele não possui uma escalabilidade forte.

A lei de Amdahl diz, de maneira resumida, que, a não ser que todo um programa serial possa ser paralelizado, o Speedup possível será bem limitado, independente do número de núcleos disponíveis. Podemos observar isso com o nosso projeto, onde, mesmo que aumentássemos a quantidade de threads, o Speedup começou a diminuir o aumento, e, com 16 threads, ficou menor do que com 8 threads. Podemos presumir que parte da culpa seja do Overhead, que sempre vai existir, e assim limitando o Speedup, mas também por conta de como o Mutex funciona.

Quando vamos atualizar uma variável global que todas as threads estão trabalhando, precisamos verificar se a variável está livre, ou seja, ninguém está alterando ela, e bloquear qualquer escrita, para não termos problema de duas threads alterarem ao mesmo tempo. Depois de bloquearmos, escrevemos, e depois ela é desbloqueada. Esse processo pode causar lentidão no código, já que as threads precisam esperar terminar a escrita para poderem continuar seu trabalho, e quanto mais threads estão sendo utilizadas, mais tempo de espera existe, dessa maneira podendo causar uma maior lentidão na execução.