Atividade 3a - Programando em Paralelo: *Pthreads* aplicado em processamento de dados

Gabriel Thiago Henrique Dos Santos <ra107774@uem.br

***Abstract*—O estudo de paralelismo tende a ser importante para diversos cenários, como em momentos nos quais há a necessidade e possibilidade de executar o mesmo serviço com "n" núcleos, no nosso caso é utilizado para o processamento de tarefas. Para a análise dos resultados e impactos, para cinco arquivos de entrada e juntamente analisado tanto para o código sequencial quanto com uso de *threads*, no caso de uma a quatro. Para melhores análises foram utilizadas da ferramenta *Performance Counters for Linux* (*Perf*) para uma melhor consulta.**

***Index Terms*—Latex template, IEEE, Latin America Transactions, guidelines for authors.**

I. Introdução

EM diversas aplicações, o uso de execuções simultâneas podem ser aplicados, e com isso se tem o objetivo principal de melhor desempenho e um período de tempo menor, para este cenário há a programação concorrente e o paralelismo, sendo diferenciadas na definição, a concorrente pode ter ser visto pelo software, em que cada bloco de execução em questão estaria concorrendo para utilizar dos recursos, já o paralelismo seria mais na perspectiva do hardware, no qual se tem os recursos para executar os bloco com "n" núcleos.

Para este trabalho implementamos para o problema com o uso das bibliotecas *POSIX Pthread* padrão, e realizamos no decorrer uma análise através de seus tempo de execução.

O seguinte relatório está organizado por sessões, no qual a II descreve sobre o problema e seus resultados esperados, em seguida é abordado sobre o desenvolvimento e suas devidas explicações. Em III há a abordagem realizada para em seguida houve a análise sobre os resultados. Ao final, temos a conclusão seguido pelas referencias utilizadas para o desenvolvimento do trabalho.

II. Objetivos

O principal objetivo deste trabalho é ganhar experiência com paralelismo multicore usando a bibleoteca *POSIX* Pthreads padrão, aplicando também os conceitos de *multithreading*, como *mutexes* e condições. Como objetivos secundários, têm-se conhecimento e experiência na implementação de um modelo de fila de tarefas mestre/trabalhador de computação paralela.

III. Descrição do Problema

O problema se baseia na leitura de uma lista de "tarefas", de um arquivo, em que cada tarefa consiste em um código de carácter, que indica uma ação, e um número que representa o tempo que esta ação é exigida.

O código "p" faz referencia à "processar", sendo o processamento de uma tarefa, e então o número n na frente significa o tempo que deve-se esperar, utilizando da função *sleep*.

Para o código "e", de "esperar", faz referencia a simulação de uma pausa nas tarefas de entrada, sendo então o número n a frente com o tempo de espera.

Deve-se também realizar o armazenamento de algumas variáveis agregadas, sendo a soma do tempo de todos os processamentos; a contagem de números de tempo impares; e o tempo mínimo e máximo dos processamento;

IV. Desenvolvimento

Utilizamos da linguagem C para nossos arquivos, para o modo sequencial houve a disponibilização juntamente com as especificações, então fora desenvolvido o com uso do paralelismo, a seguir apresentamos algumas escolhas e estruturas utilizadas no código.

- `Fila de tarefas`: criado uma fila dinâmica *queue q* com o tipo *type queue* para armazenas as tarefas a serem processadas;
- `Variável` de condição: para a variável de condição, definimos a nossa fila de tarefas *queue q*, ou seja, para utilizamos para que haja a espera das *thread* enquanto não houver tarefas nesta lista, ocasionando a espera *wait* da própria;
- `Macro`: para a melhora do processamento foram selecionados as funções de criação da fila, verificação de fila vazia e alteração da variável global com uso de *mutex lock/unlock*, e definidas ao inicio do código, com o uso do *#define*, o pré processamento não ocupa memória da execução e as referencias é alterada na chamada de cada.

Para além dessas, fora utilizado laços de repetição na função de uso das *thread*, sendo separados em laço externo e inteiro:

- `Externo`: enquanto houver serviços na lista de tarefas **OU** a *flag* de final de leitura do arquivo não ter sido setada;
- `Interno`: enquanto não houver serviços na lista de tarefas **E** a *flag* de final de leitura do arquivo não ter sido setada. Deixando então a *thread* em questão no aguardo para novos serviços.

Na função *main*, a execução pode ser vista como a Mestre, pois a mesma verifica no arquivo de entrada o comando e a ação e é inserida na lista de tarefa caso seja processamento, após a inserção a mesma acorda, com uso do *broadcast*, as *thread* ociosas para executarem o serviço.

Para a compilação do trabalho:

1) Vá a raiz do projeto e execute;

2) *make* <tipo>;

Temos para o tipo:

- sequencial
- paralelo - completar
- clear

A seguir, padronizamos para passagem dos argumentos ao momento de execução, tendo então:

- `-t:` Quantidade de *thread*, sendo as threads os trabalhadores para o processamento;
- `-f:` caminho do arquivo com a lista de serviços a serem processados;

V. Metodologia

Fora criados cinco arquivos de entrada, com as tarefas para os testes, diversificando as ações e valores. Para estudo fora definido para o primeiro arquivo vinte linhas de tarefas, o segundo com quarenta linhas, o terceiro com oitenta, o quarto com cento e sessenta e o quinto com trezentos e duzentos, ou seja, duplicando os valores. Com estes arquivos foram analisado dois quesitos, o primeiro o tempo de execução, juntamente com a resposta correta, ambos comparadas com o do código sequencial, executando então com uma, duas, três e quatro *threads*, seguindo a noção de apenas *threads* físicas do processador. E o segundo quesito analisando o desempenho com a ferramente de análise *Perf (Performance Counters for Linux)*, para este então fora selecionado apenas o código sequencial e os de duas e quatro *threads*, para todos os arquivos de entradas, para o sequencial a média de duas execuções, já para os paralelos, as médias de cinco execuções.

TABLE I: Configurações do notebook

S.O	Debian 11
Processador	AMD Ryzen 5-3500U CPU @ 2.1GHz x 4 - 64 bits
Memória RAM	8 GB's
Memória SSD	290 GB's
Placa de Vídeo	AMD Radeon Rx Vega 8

VI. Resultados

Como análises, fora decidido a execução de casos de testes a seguir para a quantidade x de *threads*, se tendo resultados pertinentes para se notar o impacto e corretude do mesmo.

A seguir, na tabela [x] se mantém os resultados do primeiro quesito de teste.

TABLE II: Resultados entrada 1

Tipo	Tempo	Resposta
Sequencial	2m29,003s	107 9 1 30
1 trabalhador	1m47,003s	107 9 1 30
2 trabalhador	1m7,002s	107 9 1 30
3 trabalhador	0m59,003s	107 9 1 30
4 trabalhador	0m52,002s	107 9 1 30

TABLE III: Resultados entrada 3

Tipo	Tempo	Resposta
Sequencial	3m56,008s	219 17 2 30
1 trabalhador	3m39,005s	219 17 2 30
2 trabalhador	1m53,004s	219 17 2 30
3 trabalhador	1m17,004s	219 17 2 30
4 trabalhador	1m0,002s	219 17 2 30

TABLE IV: Resultados entrada 3

Tipo	Tempo	Resposta
Sequencial	7m40,009s	418 30 1 40
1 trabalhador	6m58,019s	418 30 1 40
2 trabalhador	3m43,007s	418 30 1 40
3 trabalhador	2m38,004s	418 30 1 40
4 trabalhador	2m7,004s	418 30 1 40

TABLE V: Resultados entrada 4

Tipo	Tempo	Resposta
Sequencial	13m30,02s	733 59 1 40
1 trabalhador	12m13,015s	733 59 1 40
2 trabalhador	6m8,007s	733 59 1 40
3 trabalhador	4m6,026s	733 59 1 40
4 trabalhador	3m11,013s	733 59 1 40

TABLE VI: Resultados entrada 5

Tipo	Tempo	Resposta
Sequencial	30m34,08s	1681 143 1 40
1 trabalhador	28m1,115s	1681 143 1 40
2 trabalhador	14m4,014s	1681 143 1 40
3 trabalhador	9m25,009s	1681 143 1 40
4 trabalhador	7m11,008s	1681 143 1 40

É definida então a seguir os resultados primordiais da ferramenta *Perf*. Se baseando na análise de [1] , focaremos apenas em CPUs utilized (utilização), frontend cycles idle (ciclos ociosos na ULA), backend cycles idle (ciclos ociosos na busca de instrução), insns per cycle (IPC ou instruções por ciclo) , LL-cache hits (taxa de falta na cache L3) e time elapsed (tempo total de execução).

TABLE VII: Resultados Perf entrada 1

Métrica	Sequencial	2 Thread	4 Thread
frontend cycles idle	329.916	392.132	446.685
backend cycles idle	343.956	692.373	685.667
insns per cycle	0,38	0,48	0,48

TABLE VIII: Resultados Perf entrada 2

Métrica	Sequencial	2 Thread	4 Thread
frontend cycles idle	597.521	392.132	617.553
backend cycles idle	874.078	692.373	947.740
insns per cycle	0,42	0,35	0,42

TABLE IX: Resultados Perf entrada 3

Métrica	Sequencial	2 Thread	4 Thread
frontend cycles idle	320.438	934.733	926.133
backend cycles idle	393.344	941.924	1.139.505
insns per cycle	0,41	0,30	0,38

TABLE X: Resultados Perf entrada 4

Métrica	Sequencial	2 Thread	4 Thread
frontend cycles idle	262.892	1.581.429	2.011.510
backend cycles idle	274.466	1.467.461	1.328.544
insns per cycle	0,47	0,31	0,23

TABLE XI: Resultados Perf entrada 5

Métrica	Sequencial	2 Thread	4 Thread
frontend cycles idle	316.678	2.997.942	3.697.602
backend cycles idle	357.262	1.970.909	2.128.499
insns per cycle	0,43	0,24	0,21

VII. Análise e Discussão

Para o primeiro quesito, ocorre como esperado no tempo de execução, quanto mais *thread* menor o tempo para terminar o algoritmo, isto ocorre pelo paralelismo das tarefas, sendo distribuído o processamento para as *thread*, e o tempo consequentemente é diminuído na visão total. Isto fica mais claro no decorrer que se aumenta o arquivo de entrada, tendo entrada5 por exemplo o desempenho com quatro *thread* sendo 4,28x mais rápida que o sequencial.

Agora, nas tabelas analisadas com o perf, temos alguns critérios a serem notados, tendo a noção de que o *frontend cycles idle* são os ciclos ociosos na ULA, o *backend cycles idle* os ciclos ociosos na busca de instrução, a *insns per cycle* o IPC ou instruções por ciclo.

Percebe-se que quanto mais uso de *thread* no programa, maiores são os ciclos ociosos, isso ocorre em momentos em que a fila de tarefas está vazia, deixando *thread* sem trabalho, ocorrendo estas contagens em ambos ciclos idle, ficando cada vez mais notórios no aumento da entrada de instruções. O IPC, está diretamente relacionado aos ciclos ociosos, logo o indicio dele tendo a diminuir com a quantidade de ciclos ociosos. A questão para se mantiverem sem trabalho mesmo com uma quantidade grande de entrada ocorre por causa dos tempo de processamento das tarefas, no qual quando são grandes ocasiona em tempo de processamento para esta *thread* e uma espera no desenvolver do programa.

Ao final, podemos ver que ocorre o esperado no tempo de processamento porém os ciclos de ociosidade e IPC altos deveriam ser melhorados para ter um melhor desenvolvimento.

Conclusão

Finalizamos que o projeto colaborou na experiência e elaboração de códigos com paralelismo multi core juntamente com a biblioteca POSIX Pthread, desenvolvendo melhor os conceito aprendidos. O código em si elaborado atingiu as expectativas e há o bom desempenho quando comparado com o sequencial. Porém o mesmo poderia ser mais otimizado no seu desenvolvimento, focando análise para o mesmo e ao mesmo tempo alterar o código com o foco a ser melhorado. [2]

References

[1] L. F. W. Góes, “Identificando gargalos em openmp com a ferramenta perf,” p. 10, 2015.

[2] L. Calvin; SNYDER, “Snyder: Parallel programming,” vol. 24, 2008.