UNIVERSIDADE DO VALE DO ITAJAÍ

ALLAN DE LIMA DA SILVA

AVALIAÇÃO 2

Threads e Paralelismo

Relatório apresentado como requisito parcial para a obtenção da M1 da disciplina de Sistemas Operacionais do curso de Ciências da Computação pela Universidade do Vale do Itajaí da Escola do Mar, Ciência e Tecnologia.

Prof. Felipe Viel

Itajaí
2022

1. INTRODUÇÃO

No desenvolvimento de um sistema operacional, uma técnica muito utilizada para a realização de tarefas, é o paralelismo e a utilização de threads, conseguimos com estes conceitos, realizar mais de uma tarefa ao mesmo tempo, ou até mesmo dividir uma tarefa em partes para que seu tempo de execução diminua.

2. ENUNCIADO

Projeto 1

Realize uma implementação em C/C++ (ou Python) de uma multiplicação entre matrizes utilizando o sistema single thread e multithread, no qual o último deve ser feito usando as bibliotecas Pthread e OpenMP. Realize uma análise comparativa no quesito tempo de processamento utilizando bibliotecas como time.h (ou bilbioteca equivalente). A operação de multiplicação deve usar duas abordagens, a multiplicação matricial e a posicional, e deve ser entre, no mínimo, matrizes quadráticas de 3X3, como no exemplo apresentado:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad e \quad B = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

Multiplicação matricial:

$$AB = \begin{bmatrix} 1 \times 9 & 2 \times 6 & 3 \times 3 \\ 4 \times 8 & 5 \times 5 & 6 \times 2 \\ 7 \times 7 & 8 \times 4 & 9 \times 1 \end{bmatrix}$$

Multiplicação matricial:

$$A@B = \begin{bmatrix} 1 \times 9 & 2 \times 8 & 3 \times 7 \\ 4 \times 6 & 5 \times 5 & 6 \times 4 \\ 7 \times 3 & 8 \times 2 & 9 \times 1 \end{bmatrix}$$

Responda: Você conseguiu notar a diferença de processamento? O processamento (multiplicação) foi mais rápido com a implementação single thread ou multithread? Qual a biblioteca usada foi melhor para executar a operação? Explique os resultados obtidos.

Você é livre para implementar estratégias diferentes para conseguir processar bem como usar recursos de aceleração em hardware das bibliotecas.

Projeto 2

Realize uma implementação em C/C++ (ou linguagem de escolha) da soma de todas as posições com uma dimensão (um array 1D) utilizando o sistema single thread e multithread, no qual o último deve ser feito usando as bibliotecas Pthread e OpenMP. Varie o tamanho do vetor a ser somado em ambas as implementações e com ambas as bibliotecas. Varie os tamanhos do vetor, por exemplo de 10 até 150.000. Você também pode variar o tipo de dados, como inteiro e float. Exemplo de vetor:
A = [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0]

Responda: Você conseguiu notar a diferença de processamento? O processamento (soma) foi mais rápido com a implementação single thread ou multithread? Qual a biblioteca usada foi melhor para executar a operação? Explique os resultados obtidos.

3. IMPLEMENTAÇÃO

Timer

Como solicitado no enunciado, para que possamos ver as diferenças causadas por cada biblioteca, é necessário utilizar uma biblioteca que calcule com precisão o tempo de execução das funções. Para isso foi utilizada a biblioteca “chrono” que possui funções muito semelhantes a biblioteca “time.h” que foi mencionada também no enunciado.

Com a biblioteca “chrono” temos uma flexibilidade e precisão ótimas para realizar esses comparativos de tempo, dentro de cada código, será criada uma Class Timer, que de forma automática, consegue gravar o momento no qual é criada, até o momento que é finalizada a função que queremos medir o tempo, com o simples uso de blocos.

Para melhor entendimento, será utilizado a mesma notação de tempo em todos os códigos, que é a duração em microssegundos.

Projeto 1

Para a codificação deste projeto será utilizado a linguagem C++, que rodará em uma MV Ubuntu, acessada com auxílio da WSL disponível no Windows, o mesmo será feito para o “Projeto 2”.

Iniciamos com a resolução do programa feito sem nenhuma técnica de paralelismo, dentro da aplicação existem 4 funções para as respectivas finalidades, preenchimento de matriz, impressão de matriz, multiplicação matricial e multiplicação posicional.

Devido ao tipo de multiplicação que será efetuada, as matrizes vão sempre possuir o mesmo tamanho, que é definido por uma variável ‘TAM’, que terá seu valor alterado diversas vezes para que seja testado o código e seu tempo de execução.

O preenchimento das matrizes é feito de maneira aleatória, onde cada posição receberá um valor entre 0 a 10 com uso da função ‘rand()%10’.

Ambas as multiplicações de maneira muito semelhante, recebem duas matrizes preenchidas com valores inteiros, e uma matriz na qual será registrado o valor da multiplicação, único detalhe diferente entre as funções será o acesso a segunda matriz recebida, que terá suas colunas trocadas por suas linhas, como exemplificado no enunciado do trabalho.

```
void multiplicacao_matricial(int A[TAM][TAM], int B[TAM][TAM], int
C[TAM][TAM]){
    for(int i = 0; i <TAM ; i++){
        for(int j = 0; j < TAM ; j++){
            C[i][j] = A[i][j] * B[j][i];
        }
    }
}
```

```
void multiplicacao_posicional(int A[TAM][TAM], int B[TAM][TAM], int
D[TAM][TAM]){
    for(int i = 0; i <TAM ; i++){
        for(int j = 0; j < TAM ; j++){
            D[i][j] = A[i][j] * B[i][j];
        }
    }
}
```

A classe timer será chamada duas vezes, uma vez para cada função de multiplicação, desconsiderando o tempo de execução para o preenchimento e a apresentação dos resultados no terminal.

A implementação da biblioteca ‘OpenMp’ neste projeto foi feita utilizando o parâmetro ‘#pragma omp parallel for’ logo antes de cada início de multiplicação, para que houvesse o paralelismo

Com a implementação da biblioteca ‘Pthread’, foi definido que os melhores resultados se obtiveram ao utilizar 2 threads, separando uma função de multiplicação para cada thread.

```
    ...
    pthread_create(&threads_mult,NULL, multiplicacao_matricial,NULL);
    pthread_create(&threads_pos,NULL, multiplicacao_posicional,NULL);
    pthread_join(threads_mult,NULL);
    pthread_join(threads_pos,NULL);
    ...
```

Projeto 2

Para o iniciar a implementação deste projeto, foi feita a codificação em maneira sequencial sem nenhum tipo de paralelismo, o que acabou sendo uma tarefa muito simples, com poucas funções utilizadas e poucas variáveis também.

Fazemos duas execuções neste projeto, para melhor analisarmos os dados que serão recebidos, primeiro é feita uma função onde é criado um vetor, com o tamanho que será definido varias vezes a fim de termos vários resultados para nossa tabela, o conteúdo desse vetor é gerado de forma aleatória, podendo ter valores entre 0 a 1000, este numero é armazenado dentro do vetor, e logo é somado a uma variável que passará por todas as posições do vetor, onde após terminar o seu preenchimento, será apresentado no terminal essa variável com o valor de todas as posições do vetor somadas. Junto deste resultado, recebemos também com o auxílio da classe Timer implementada, o tempo total de duração desta função.

```
void vetor_int(){
    int vetor[TAM];
    long int vetorSum = 0;
```

```
    long int i;

    for(i = 0 ; i < TAM ; i++){
        vetor[i] = 22;
        vetorSum = vetorSum + vetor[i];
    }
    cout<<endl<< "soma do vetor de inteiros: " <<vetorSum;
}
```

A segunda execução deste projeto, será muito semelhante a primeira, com a diferença que esta função conterá um vetor de números do tipo FLOAT, esse que também são gerados aleatoriamente, com duas linhas, onde conseguimos gerar um número de 2 dígitos inteiros e 2 decimais.

```
vetor[i] = rand()%100;
vetor[i] = vetor[i]/100 + rand()%100;
```

Para a realização deste código utilizando a biblioteca OpenMp, temos uma pequena modificação feita, um parâmetro para o “for” de cada função, indicando, que ele será feito em paralelo, e também indicando que a variável soma será compartilhada, para que não haja nenhum conflito.

```
#pragma omp parallel for reduction(+:vetorSum)
```

Para o código utilizando a biblioteca “pthread”, temos mais alguns modificações, como a definição de Threads disponíveis para utilizar, que no caso serão utilizadas ‘4’, um vetor de resultados, para armazenar a soma feita por cada thread em cima do mesmo vetor, e de grande importância também, uma variável de controle, que define qual parte do

vetor será verificada por cada thread, para que o vetor seja dividido igualmente entre todas as threads.

4. RESULTADOS

Projeto 1

Para melhor analisarmos o desempenho de cada código, foram feitos diversos testes, onde cada um possui algumas mudanças significativas. Para que fosse encontrado os melhores resultados, os códigos foram modificados várias vezes, a fim de tentar achar a melhor execução possível no cenário.

	matricial	posicional	matricial	posicional	matricial	posicional
	SingleThread		OpenMP		PThread	
TAM = 5	0	0	1	0	0	0
TAM = 100	1	0	1	0	0	1
TAM = 500	6	4	7	6	4	4

***Tempo em microssegundos.**

O execuções foram feitas alternando o valor ‘TAM’ que corresponde ao tamanho das matrizes, onde conseguimos observar um maior período de tempo de execução quando usamos matrizes de tamanho 500x500, que geraram um tempo médio de execução de aproximadamente 4-5 microssegundos, levando isso em conta, podemos observar que o melhor cenário foi recebido com a execução utilizando a biblioteca ‘Pthread’ onde demorou apenas 4 microssegundos para executar ambas funções, que rodaram ao mesmo tempo pelo sistema.

E tivemos o pior cenário sendo a utilização da biblioteca ‘OpenMp’, pois sendo uma simples multiplicação, pode ter sofrido com atrasos para que os mesmos endereços de memória não fossem acessados ao mesmo tempo por diferentes threads, a fim de não gerarem resultados errados.

Projeto 2

Assim como no projeto anterior, diversos meios de execução foram testados a fim de tentar achar a melhor execução possível no cenário.

	int	float	int	float	int	float
	SingleThread		OpenMP		PThread	
TAM = 10000	1	0	1	2	2	3
TAM = 200000	4	5	2	41	16	34
TAM = 400000	5	7	3	72	33	68

***Tempo em microssegundos.**

Primeiramente foram realizados teste mudando apenas o tamanho do vetor, vemos uma diferença considerável em relação aos testes feitos com paralelismo, onde dentro desse cenário com esses possíveis tamanhos de vetor, se viu mais vantajoso o uso de um código sem paralelismo, se possível.

Com esse resultado, que não foi muito exatamente esperado, o código foi alterado, para que ao invés de gerar números aleatórios, o vetor fosse preenchido com números pré-definidos

	int	float	int	float	int	float
	SingleThread		OpenMP		PThread	
TAM = 10000	0	0	0	0	1	2
TAM = 200000	2	1	2	0	2	2
TAM = 400000	3	2	2	1	3	3

***Tempo em microssegundos.**

***Números não aleatórios.**

Com esta modificação, temos uma grande diferença vista nos resultados, onde o tempo de todas a execuções, principalmente as com paralelismo, foram decrementadas, conseguimos ver maior viabilidade na utilização destas bibliotecas neste caso.

	int	float	int	float	int	float
	SingleThread		OpenMP		PThread	
TAM = 10000	10	15	16	16	15	15
TAM = 200000	57	58	220	129	260	240
TAM = 400000	123	111	162	255	194	256

***Tempo em microssegundos.**
***Incremento de um 'cout<<'**

Um ultimo teste é feito, com a única modificação sendo uma linha com um simples, mas custoso, ' cout<<"." ', sem nenhum objetivo em específico além de causar maior peso para a simples função, vemos que essa função acaba decrementando de maneira significativa a otimização do código e novamente vemos uma disparidade entre o uso de paralelismo.
É visto que em uma aplicação consideravelmente muito simples, como esta soma de valores de um vetor, acaba não sendo vantajoso o uso de paralelismo, pois o custo de manuseio deles acaba sendo maior que o próprio custo de aplicação do código, em um cenário onde essa situação fosse de outra escala, como por exemplos dezenas ou centenas de vetores, provavelmente conseguiríamos observar a predominância de uma aplicação com paralelismo.

5. CONCLUSÃO

Após realizar os projetos apresentados no enunciado, conseguimos observar com clareza o funcionamento do paralelismo, com seus benefícios e suas dificuldades, que podem ser vistas pela complexidade aumentada do código, principalmente utilizando a biblioteca Pthreads, e também um aumento não significativo ou até mesmo inexistente na realização de tarefas muito simples.