Cálculo de π usando threads

Gabriel Gaspar, Pedro Olivo

03 dezembro 2024

1 Introdução

Trabalho realizado pelos alunos Gabriel Gaspar e Pedro Olivo, para a matéria de Arquitetura e Sistemas Operacionais, ministrada pelo professor Gabriel Candido. Neste trabalho, foi feito um programa em C++, para calcular os termos de π, usando threads. Os programas foram compilados usando o seguinte comando, no qual foi solicitado na atividade:

```
g++ -o calcpi calcpi.cpp -lpthread
```

2 Ambiente de Desenvolvimento

Os programas foram desenvolvidos e testados em um computador Samsung Galaxy Book 4 Pro, utilizando o Windows Subsystem for Linux (WSL), na edição Linux 5.15.153.1-microsoft-standard-WSL2, usando como sistema operacional convidado o Ubuntu 22.04.5 LTS, com as seguintes especificações:

- Sistema Operacional: Windows 11 / Ubuntu 22.04.5 LTS
- Processador: Intel Core Ultra 7 155H
- Memória: 16GB DDR5 7467MT/s
- Armazenamento: SAMSUNG MZVL4512HBLU-00B
- IDE: Visual Studio Code 1.95.3

3 Série de Leibniz

Ao implementar a série de Leibniz em C++, podemos convertê-la do formato de somatório a seguir:

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4}$$

para código em C++:

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    double pi = 0;
    for (long long int n = 0; n < 1000000000; ++n) {
        pi += (pow(-1, n) / (2 * n + 1)) * 4;
    }
    cout << "Resultado: " << pi << endl;
}
```

Este código, de acordo com a atividade, calcula um bilhão (10^9) de termos da série de Leibniz, usando variáveis de precisão dupla e também usando a biblioteca **cmath**, que é o equivalente ao **math.h** , do C, para o C++.

4 Cálculo de π usando várias tarefas

De acordo com a atividade, o código do capítulo anterior foi alterado para fazer tal cálculo usando tarefas paralelas, usando threads. Para poder separar o código anterior em várias tarefas, foi feita a função **calcpi**, que separa uniformemente o espaço de cálculo entre a quantidade especificada no início do código de tarefas. Na função **main**, está a criação de threads, ainda de acordo com a quantidade especificada e a chamada da função. No final, todos os valores achados por cada tarefa são alocados no vetor global para que todos os seus termos sejam somados, assim encontrando a resposta do cálculo de π, com uma precisão de 10 dígitos decimais. Não queríamos usar o vetor global que foi usado, mas era necessário para não mexer com alocação dinâmica de memória. Com isso, segue o código do programa em C++:

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>
#include <pthread.h>
using namespace std;

#define NUM_THREADS 2

double results[NUM_THREADS];

void *calcpi(void *threadid) {
    long id = (long)threadid;
    double soma = 0;

    long long inicio = id * (1000000000 / NUM_THREADS);
    long long fim = (id + 1) * (1000000000 / NUM_THREADS);

    for (long long n = inicio; n < fim; ++n) {
        soma += (pow(-1, n) / (2 * n + 1)) * 4;
    }

    results[id] = soma;
    pthread_exit(nullptr);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int status;

    for (long i = 0; i < NUM_THREADS; i++) {
        cout << "Criando thread " << i << endl;
        status = pthread_create(&threads[i], nullptr, calcpi, (void *)i);

        if (status) {
            cerr << "Erro ao criar thread " << i << endl;
            exit(-1);
        }
    }

    for (long i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], nullptr);
    }

    double pi = 0;
    for (int i = 0; i < NUM_THREADS; i++) {
        pi += results[i];
    }

    cout << fixed << setprecision(10) << "Resultado: " << pi << endl;

    return 0;
}
```

5 Tempo por quantidade de threads

Foi modificado o código anterior, para poder calcular o tempo, utilizando a biblioteca **chrono**, além de colocar todo o método main dentro de um laço *for*, para que o código rode 5 vezes automaticamente. No final, foi implementada uma soma para que a média do tempo demorado seja feita também automaticamente, além de ter sido criada uma saída que imprime as coordenadas que são relevantes para a confecção de gráficos, facilitando o desenvolvimento dos mesmos em

LaTeX. O código está abaixo, de modo em que a linha 8 foi alterada conforme a quantidade de threads solicitada pela atividade.

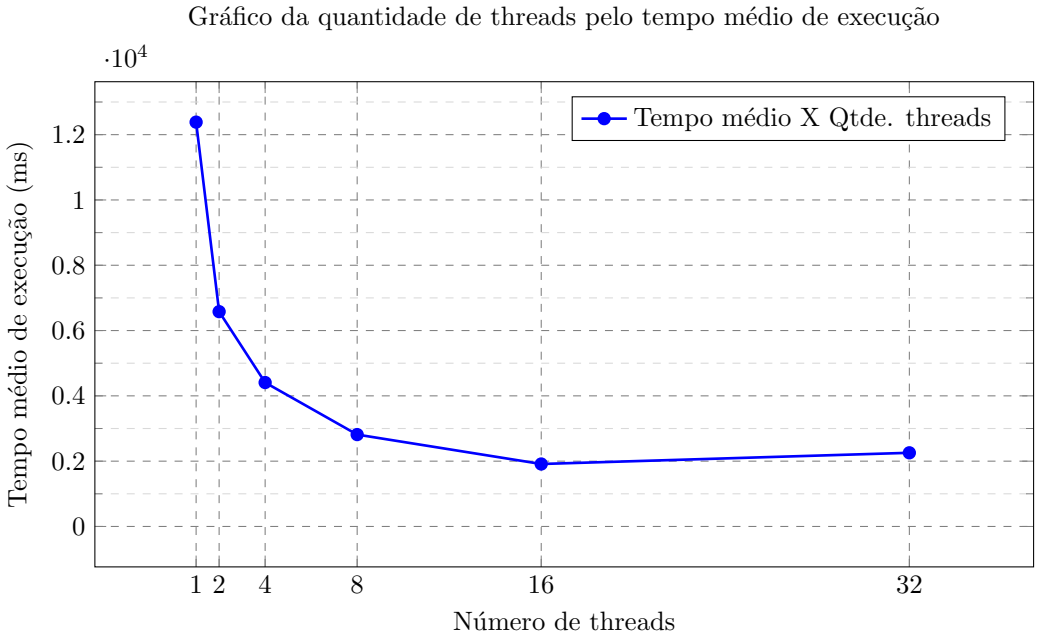
```cpp
#include <iostream>
#include <iomanip>
#include <cmath>
#include <pthread.h>
#include <chrono>
using namespace std;

#define NUM_THREADS 1

double results[NUM_THREADS];

void *calcpi(void *threadid) {
    long id = (long)threadid;
    double soma = 0;

    long long inicio = id * (1000000000 / NUM_THREADS);
    long long fim = (id + 1) * (1000000000 / NUM_THREADS);

    for (long long n = inicio; n < fim; ++n) {
        soma += (pow(-1, n) / (2 * n + 1)) * 4;
    }

    results[id] = soma;
    pthread_exit(nullptr);
}

int main() {
    int media = 0;
    for (int vezes = 1; vezes <= 5; vezes++) {
    auto start = std::chrono::high_resolution_clock::now();
    pthread_t threads[NUM_THREADS];
    int status;

    for (long i = 0; i < NUM_THREADS; i++) {
        //cout << "Criando thread " << i << endl;
        status = pthread_create(&threads[i], nullptr, calcpi, (void *)i);

        if (status) {
            cerr << "Erro ao criar thread " << i << endl;
            exit(-1);
        }
    }

    for (long i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], nullptr);
    }

    double pi = 0;
    for (int i = 0; i < NUM_THREADS; i++) {
        pi += results[i];
    }

    //cout << fixed << setprecision(10) << "Resultado: " << pi << endl;

    auto stop = std::chrono::high_resolution_clock::now();

    auto d1 = std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);

    cout << "(" << vezes << ", " << d1.count() << ") ";

    media += d1.count();
    }
    cout << endl;
    cout << "media: " << media / 5 << endl;
    return 0;
}
```

Abaixo está uma tabela com os tempos de execução obtidos, valores médios e coeficientes de variação, específicos para cada quantidade de thread solicitada. Os valores médios são a média dos tempos de execução e o coeficiente de variação

é para garantir que as medições estejam consistentes, usando sua própria fórmula. Todas as medidas de tempo estão em milissegundos (ms), por motivos de padronização e para facilitar os cálculos da média e dos coeficientes de variação.

	Tempo de execução (ms)	Valores médios (ms)	Coeficientes de variação (%)
1 thread	11999; 12304; 12330; 12614; 12672	12383	1,78%
2 threads	6685; 6318; 6646; 6693; 6555	6579	2,12%
4 threads	4265; 4329; 4623; 4509; 4314	4408	3,07%
8 threads	2664; 2822; 2747; 2893; 2948	2814	3,59%
16 threads	1816; 1826; 1924; 1936; 2065	1913	4,71%
32 threads	2136; 2218; 2355; 2360; 2211	2256	3,88%

6 Gráfico de threads X tempo



7 Conclusão

Com todas as informações supracitadas, é possível notar que quanto mais threads se usam para executar um programa, mais rápido ele vai terminar de executar. Por exemplo, com somente uma thread, o programa demorou, em média, 12 segundos para terminar de rodar, enquanto com 32 threads, o programa demorou aproximadamente 10x menos, com um total de 2 segundos. Resumindo, para operações que demandam muito do sistema, é extremamente conveniente utilizar threads, já que cada thread vai executar uma parte do programa em paralelo e de modo simultâneo, podendo reduzir em grandes quantidades o tempo necessário para que o programa termine de rodar.

Por fim, é importante notar também, que ao rodar o programa usando o comando **time**, do Linux, mostra que o programa com 32 threads demorou 8 segundos para rodar totalmente, mesmo tendo 3 minutos e 5 segundos de tempo de usuário. Isso se dá porque o sistema operacional calcula o tempo que cada thread demorou no modo de usuário/sistema do processador e soma todos os valores no final, para mostrar como tempo real, diferente das medições feitas anteriormente, que usam as funções da biblioteca **chrono**, do próprio C++, as quais só levam em conta o tempo real.