Análise do algoritmo de multiplicação de matrizes sequencial e paralelo

Felipe Diniz Tomás - RA:110752

***Abstract*—O presente estudo realiza uma análise de um algoritmo de multiplicação de matrizes implementado de forma paralela e sequencial, com o objetivo de analisar e mensurar o desempenho do código paralelo. Sendo assim foi coletado o tempo de execução de determinados tamanhos matriciais com intuito de calcular o *speedup*[1] dos mesmos. Os resultados de *speedup* não se mostraram ruins considerando o hardware utilizado, chegando próximo ao linear em entradas maiores testadas, consequentemente trazendo uma redução significativa de tempo.**

I. Introdução

A multiplicação de matrizes é uma operação importante em muitos algoritmos numéricos, contudo tal problema executado de forma tradicional exige um alto custo de recursos, tanto de processamento como principalmente de tempo. Aplicando diretamente a definição matemática de multiplicação de matrizes, sabe-se que o algoritmo está na ordem de $O(n^3)$, o que é considerado algo longe do ideal.

Muito trabalho foi investido para tornar os algoritmos de multiplicação de matrizes eficientes, uma forma encontrada de contornar o problema é o uso de hardwares específicos que permitem a implementação de um sistema paralelo. Na computação paralela, uma tarefa computacional é normalmente dividida em várias subtarefas muito semelhantes, que podem ser processadas de forma independente e cujos resultados são combinados posteriormente, após a conclusão.

Tendo ciência disso, uma ferramenta muito difundida que tem como objetivo criar e manipular threads é a POSIX threads, a qual define uma API padrão para controle das mesmas. As bibliotecas que implementam a POSIX threads são chamadas *Pthreads* e trazem recursos necessários para aplicação da computação em paralelo.

Portanto, o objetivo do trabalho é realizar a implementação do código sequencial e paralelo de multiplicações de matrizes com a utilização da biblioteca *Pthreads*, produzindo uma análise da métrica de *speedup* a fim de esclarecer os resultados encontrados e estabelecer um padrão de análise de código.

II. Metodologia

As execuções realizadas foram feitas no seguinte ambiente computacional:

- Processador Intel® Core™ i5-4210U
  - Número de núcleos: 2;
  - Número de threads: 4;
  - Frequência baseada em processador: 1.70 GHz;
  - Frequência turbo max[2]: 2.70 GHz;
  - Cache: 3 MB Intel® Smart Cache[3]
    - Tamanho da memória cache:
      - L1 - 128 KB;
      - L2 - 512 KB;
      - L3 - 3 MB.
- 8 Gb de memória RAM DDR3
- Sistema Operacional: Windows 10 utilizando o Subsistema Windows para Linux (WSL)[4] 2.0, sendo ele:
  - Ubuntu 21.04 LTS;
  - Kernel: 5.10.16.3;
  - Compilador: gcc 9.3.0.

Os casos de testes realizados incluiam matrizes de tamanho 128, 512, 1024 e 2048, sendo executados de forma sequencial e paralela. Para cada tamanho foi realizado um total de 6 testes, sendo o primeiro deles descartado para evitar *compulsory miss*[5], e calculado a média dos demais.

III. Resultados

Os resultados de tempo do código sequencial para as suas respectivas entradas foram:

TABLE I: Tempo em segundos do código sequencial.

128x128	512x512	1024x1024	2048x2048
0,025s	0,998s	14,833s	198,751s

Os resultados de tempo do código paralelo ara as suas respectivas entradas foram:

TABLE II: Tempo em segundos do código paralelo.

Threads	128x128	512x512	1024x1024	2048x2048
2	0,018s	0,611s	7,502s	105,074s
4	0,0165s	0,517s	5,365s	55,179s
8	0,015s	0,498s	4,542s	44,156s

[1]O *speedup* é um valor que mede a melhoria na velocidade de execução de um processo executado em duas arquiteturas semelhantes com recursos distintos.

[2]Frequência turbo máxima é a frequência máxima de núcleo único, à qual o processador pode funcionar, usando a Tecnologia Intel® Turbo Boost.

[3]Cache inteligente refere-se à arquitetura que permite que todos os núcleos compartilhem dinamicamente o acesso ao cache de último nível.

[4]Windows Subsystem for Linux permite executar qualquer binário compilado para Linux diretamente no Windows, acompanhando inclusive um próprio kernel Linux

[5]O *compulsory miss* é uma falha que ocorre quando um bloco é trazido pela primeira vez a memória cache.

Calculado o tempo para cada tamanho, tanto do código paralelo como sequencial, é aplicado a fórmula abaixo para o calculo de *Speedup* para cada tamanho matricial.

$$S(t) = \frac{Tempo\ de\ execução\ Sequencial}{Tempo\ de\ execução\ com\ t\ Threads}$$

A figura 1 a seguir indica o *Speedup* alcançado para cada tamanho matricial processados pelo algoritmo de multiplicação de matrizes paralelizado.

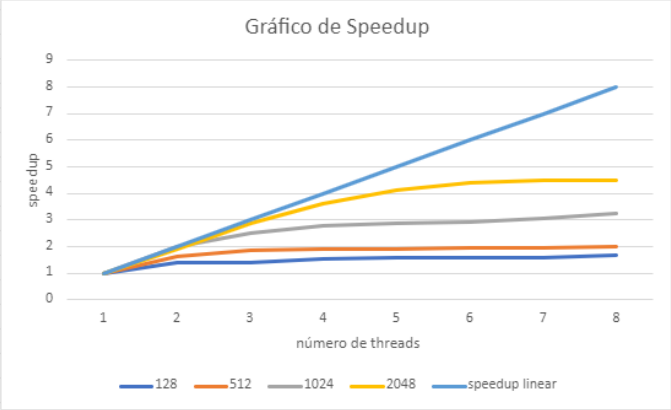

Fig. 1: Gráfico de *Speedup* do problema de multiplicação matricial.

Primeiramente vale ressaltar que os testes feitos em um processador de apenas 2 núcleos físicos tem resultados mais próximos ao linear quando usado os dois núcleos físicos, a partir daí serão usados núcleos virtuais e consequentemente o *Speedup* não tem a mesma escala com núcleos virtuais do que com núcleos físicos. O que acontece é que processador não tem recursos para executar simultaneamente dois fluxos distindos, então ele escalona instruções das *threads* de fluxos diferentes em um mesmo núcleo físico, logo a queda de tempo menos acentuada é algo esperado. Isso explica o motivo do *Speedup* se distanciar do linear com a utilização de mais de 2 *threads*.

Observa-se que para entradas menores como 128 e 512, há um *Speedup* que não acompanha o ideal linear com duas *threads*. Para explicar tal situação é necessário entender que o problema de multiplicação matricial é de paralelismo de dados, ou seja, todas as *threads* executam o mesmo conjunto de instruções porém processam parte diferente dos dados, logo pode acontecer da quantidade de tarefas para cada *thread* não ser suficiente, sabendo disso deve-se aumentar o tamanho da entrada e observar se o problema persiste, o que nesse caso não ocorreu.

Entradas maiores de 1024 e 2048 o *Speedup* se aproximou mais ao ideal com duas *threads*, chegando a 1,97 e 1,89 respectivamente. Mesmo pela falta de alguns décimos comparado ao linear, ainda é um resultado satisfatório, pois essa diferença pode ser causada por vários fatores externos ao código, como por exemplo o próprio processador, sitema operacional, etc.

Constata-se também que não houve um *Speedup superlinear*, que a pesar de serem raros, podem ocorrer por exemplo ao realizar algoritmos de *Backtracking* de forma paralela [1]. Já *slowdown* também não ocorreu, geralmente são causados por um gargalo de comunicação, neste caso o problema de multiplicação de matrizes é embaraçosamente paralelo[6], logo, não necessitam de tal comunicação.

IV. Conclusão

É notável que a abordagem paralela para aplicação do problema de multiplicação de matrizes torna a execução consideravelmente mais rápida, assim quando se quer um resultado ótimo, utilizar e analisar a métrica de *Speedup* é algo essencial.

No entanto uma análise de *Speedup* pode se tornar um tanto quanto complexa, diversos fatores podem influenciar o desempenho do programa, e torná-lo ideal é um tanto desafiador. O trabalho possibilitou uma compreensão melhor do que pode causar tais quedas de desempenho e como trabalhar com a programação paralela afim de conseguir bons resultados.

Considerando projetos futuros, seria interessante a utilização de um processador mais moderno, com mais núcleos físicos, afim de estudar o comportamento do código, também seria importante utilizar entradas maiores para uma analise de desvio do *Speedup* mais precisa, quanto ao código, é necessário um estudo melhor com ferramentas afim de reduzir seu tempo de execução em busca de alcançar uma performance mais próxima ao *Speedup* linear.

References

[1] E. Speckenmeyer, "Superlinear speedup for parallel backtracking," *Lecture Notes in Computer Science*, no. 297, p. 985–993, 2005.
[2] I. Foster, "Designing and building parallel programs," *Addison–Wesley*, 1995.

[6]Um problema embaraçosamente paralelo é aquele que há pouca ou nenhuma dependência ou necessidade de comunicação entre as tarefas paralelas ou dos resultados entre elas [2].