Verificação da paralelização da soma harmônica com OpenMP e CUDA

Eugênio A. K. Nishimiya[1], Nataly C. da Silva[1], Hélio C. Guardia[1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – 13565-905 – São Carlos – SP – Brasil

{eugenio, nataly.cristina}@estudante.ufscar.br,helio.guardia@ufscar.br

Abstract. *This paper investigates the efficiency for parallel solutions for the sum of the n first numbers of an harmonic progression. To this end, we consider an implementation using OpenMP, which considers shared memory, and another using CUDA, which leverages GPU processing. The results are evaluated based on metrics such as speedup and efficiency, providing a clearer understanding of the time gains produced through parallelization.*

Resumo. *Este artigo busca verificar a eficiência de soluções para a paralelização da soma dos n primeiros termos de uma progressão harmônica. Para tanto, consideramos uma implementação em OpenMP, que utiliza memória compartilhada, e outra em CUDA, que utiliza o processamento em GPU. Os resultados foram avaliados em função de métricas como speedup e eficiência, que mostram mais nitidamente os ganhos de tempo com a paralelização da aplicação.*

1. Introdução

A soma harmônica é a soma dos termos de uma progressão harmônica especial, onde são tomados os recíprocos dos números naturais. O problema reside na inexistência de uma fórmula fechada para o cálculo da soma dos *n* primeiros termos dessa progressão, sendo que cada termo deve ser computado e somado com os demais, o que pode ser custoso em termos de tempo. Sua investigação se justifica na medida em que essa soma é utilizada em matemática, nos estudos sobre teoria de números, e em física, em campos como mecânica quântica.

O presente estudo investiga as técnicas de paralelização para a solução desse problema. Para tanto, utilizamos da biblioteca OpenMP, que permite a paralelização do código em um ambiente de memória compartilhada, e CUDA, que fornece os meios para a paralelização utilizando recursos da GPU. Como referência, utilizamos do código sequencial desse problema fornecido pela 16ª Maratona de Programação Paralela [Bianchini and Pillon, 2021]. Os códigos desenvolvidos podem ser encontrados publicamente em um repositório do Github[1].

2. Soma dos termos da progressão harmônica

O algoritmo sequencial fornecido, mostrado no Algoritmo 1, trata-se de uma inicialização de um vetor de caracteres, onde cada índice representa um dígito e o seu tamanho é dado pela precisão em casas decimais fornecida através de *input*. Os termos são calculados com base em somas parciais do resto da divisão de dígitos anteriores. Como se trata de um laço aninhado, onde o *loop* exterior opera sobre os *n* primeiros

[1] https://github.com/compermane/parallel-harmonic-sum

termos da progressão harmônica e o laço interior opera sobre os *d* dígitos de precisão, podemos dizer que esse algoritmo tem uma complexidade de O(*n* * *d*).

Algoritmo 1 Cálculo da Soma Harmônica

```
1: Entrada: vetor de resutado v1, precisão d, limite superior da soma n
2: iniciar v2 com tamanho d                                  ▷ vetor de dígitos
3: para cada dígito em v2 faça
4:     digito ← 0
5: fim para
6: para i = 1 até n faça
7:     resto ← 1
8:     para j = 0 até d e resto ≠ 0 faça
9:             quociente ← resto/i
10:            resto ← (resto mod i) × 10
11:            v2[j] = v2[j] + quociente
12:    fim para
13: fim para
14: v1 ← v2
15: Saída: Soma harmônica em v1
```

A principal estratégia de paralelismo neste problema consiste em atribuir às *threads* intervalos diferentes de *n,* onde cada linha de execução realiza as somas parciais dentro desse intervalo e, posteriormente, as somas parciais são sincronizadas de forma a obter o resultado final [Kaur, Kumar and Patle, 2014].

3. Metodologia

Para paralelizar o problema descrito, utilizamos o *framework* OpenMP e o ambiente de GPU fornecido pela plataforma CUDA. Para a implementação da solução com OpenMP, utilizamos um ambiente com o processador Intel(R) Xeon(R) CPU E7-4870, contando com 256GB de RAM, 4 *sockets*, 10 núcleos por *socket* e 2 *threads* por núcleo. Já para a paralelização com CUDA, utilizamos o ambiente fornecido pelo Google Colaboratory que conta com uma GPU Tesla T4, tendo 16GB de memória VRAM.

3.1. Paralelização utilizando OpenMP

A principal estratégia utilizada para a paralelização utilizando OpenMP foi a distribuição de intervalos de *n* do laço externo para cada *thread*. Nesse caso, foi paralelizado o laço externo do Algoritmo 1. Para evitar condições de corrida durante a agregação dos resultados parciais, utilizou-se a cláusula *reduction*. Essa abordagem permite que cada *thread* mantenha sua própria instância local do vetor de soma, acumulando os resultados de forma independente. Ao final da execução paralela, OpenMP realiza a operação de redução, combinando os vetores locais elemento a elemento, ou seja, somando cada posição correspondente dos vetores de forma alinhada.

Na versão paralela, a execução idealmente é dividida entre *p threads*, resultando em uma complexidade aproximada de O((*n* * *d*) / *p*), onde *p* é o número de *threads*.

3.2. Paralelização utilizando CUDA

Para a solução com CUDA, atribuímos a um *kernel* o processamento de um intervalo do vetor inicializado, ou seja, o trecho paralelizado também é o laço externo do Algoritmo

1. Nesse caso, cada *thread* processa um valor baseado em seu identificador, acumulando a soma calculada em uma posição de memória compartilhada entre as *threads.* Como há o uso de memória compartilhada, devemos utilizar operações atômicas para evitar condições de corrida, tendo em vista que múltiplas *threads* podem acessar uma mesma posição de memória ou atualizar um mesmo dígito, além de utilizar métodos de sincronização de *threads*. Assim sendo, cada *thread* faz as operações de somas atômicas em função de uma posição do vetor de dígitos, atualizando cada dígito atomicamente.

Como pode-se verificar na Tabela 1, para essa abordagem, o *overhead* de comunicação entre CPU e GPU é mínimo, sendo quase todo o tempo de GPU utilizado pelo *kernel*. Esses resultados foram obtidos através do perfilamento do código com **nvprof**. Para as outras configurações, foram obtidos resultados semelhantes.

Tabela 1. Relação entre o *overhead* de comunicação entre CPU e GPU

Threads por bloco	Tamanho do problema	Tempo de Comunicação (µs)	Tempo total de GPU (µs)	Tempo de GPU (%)
128	100000	3.711	2467.411	0.15
128	1000000	3.393	2860.393	0.02
128	10000000	2.240	83784.24	0.002

4. Resultados e Discussão

Nesta seção, analisamos os resultados obtidos a partir da paralelização do cálculo da soma harmônica utilizando as duas abordagens exploradas. Para fins de comparação, usamos precisão de 256 dígitos e tamanhos de problema com n=100000, n=1000000 e n=10000000, com tempos sequenciais de 0,582s, 5,975s e 60,5s, respectivamente.

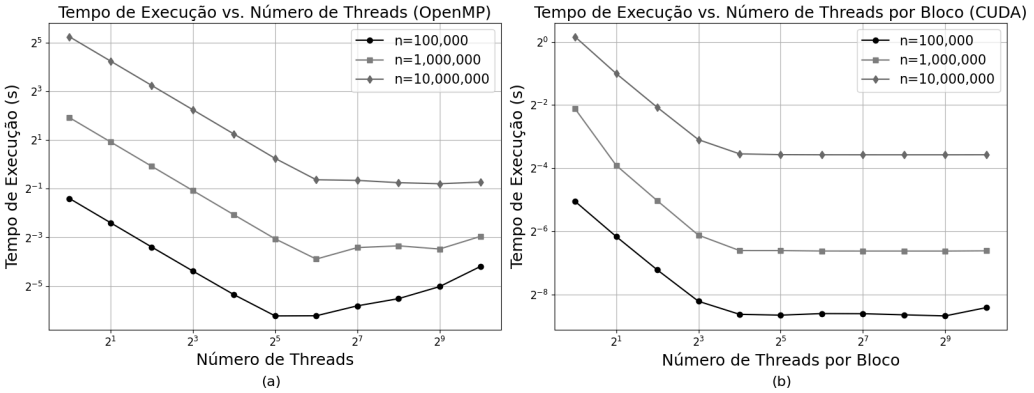

Figura 1. Tempos de execução vs. quantidade de *threads*

Para ambas as abordagens, os tempos de execução diminuem até certo ponto, estabilizando-se em seguida. No caso do CUDA (Figura 1b), o menor tempo foi obtido com 16 *threads* por bloco. A execução paralela é estruturada em grades compostas por blocos, e cada bloco possui um número fixo de *threads*. O total de blocos é obtido pela

fórmula (*n* + *n_threads* - 1) / *n_threads*, garantindo que cada termo soma seja processado por uma *thread*. O número total de *threads* é aproximadamente *n*, distribuídas entre os blocos. Como a GPU executa vários blocos simultaneamente, o grau de paralelismo é alto, mas é limitado pelos seus recursos, o que explica a posterior estabilização de desempenho

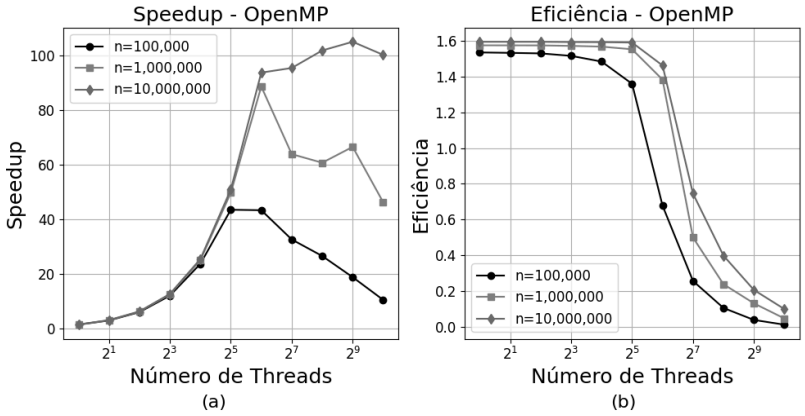

Figura 2. Gráficos de s*peedup* e de eficiência para a solução com OpenMP.

Para a solução com OpenMP, onde os menores tempos de execução foram em torno de 32 e 64 *threads* (Figura 1a), podemos notar que o *speedup* (Figura 2a) e a eficiência (Figura 2b) alcançam seus valores máximos em torno de 64 *threads*, com exceção da execução com n=10000000, que apresentou *speedup* crescente até 512 *threads* e a eficiência que chegou a valores superlineares até 32 *threads*. Esse comportamento coincide com a quantidade de *threads* que o *hardware* utilizado para os experimentos consegue comportar, indicando que todos os recursos computacionais são utilizados. Conforme se aumenta o número de *threads* e o tamanho do problema, o *overhead* de sincronização também cresce, justificando a piora no desempenho.

5. Conclusões

Em suma, podemos notar que as soluções paralelas para o problema da soma dos primeiros termos de uma progressão harmônica reduzem drasticamente o tempo de execução em relação a uma abordagem sequencial. No entanto, deve-se ter cuidado ao utilizar tais técnicas, tendo em vista que podem ocorrer condições de corrida, acarretando em um resultado equivocado. Para resolver essas situações, utilizamos da operação de redução em OpenMP e somas atômicas na implementação com CUDA.

References

Bianchini, C. and Pillon, M. A. (2021). 16th marathon of parallel programming: Rules for remote contest. In Proceedings of the 16th Marathon of Parallel Programming, pages 1–1, Brazil. SBAC-PAD & WSCAD.

Kaur, R., Kumar, S. and Patle, K., 2014, A Parallel Algorithm to Process Harmonic Progression Series Using OpenMP, INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT) Volume 03, Issue 02 (February 2014).