Universidade de São Paulo
Instituto de Física de São Carlos

SFI5822 - Introdução à Programação Paralela (2024)
Prof. Gonzalo Travieso

Trabalho A - Paralelo

Lucas Constante Mosquini - **NºUSP** 11858258

December 17, 2024

Contents

1 Introdução

Como descrito no projeto, temos que fazer um programa que recebe as coordenadas de N partículas, em um espaço tridimensional, linha por linha. Em seguida, temos que calcular a distância euclidiana entre elas e assim, analisar os m vizinhos mais próximos. Por fim, imprimi-se um arquivo, com as informações N, m, tempo de leitura de entrada, tempo de cálculo e tempo de escrita.

Como diretrizes principais: respeitar boas práticas, como visto em aula e especialmente pensar em código que possa ser paralelizado facilmente futuramente, ou seja, evitar usar estruturas complexas como árvores de armazenamento, mesmo que elas tenham um desempenho superior.

Por fim, esse pdf se resume em justificar as escolhas visando o desempenho, assim como explicar o funcionamento e ideias bases por trás do código.

Dessa vez, a proposta vai ser otimizar o código sequencial, paralelizando-o usando o MPI.

2 Proposta de Paralelismo

Nesta secção, será discutida a forma como foi feito o esqueleto do projeto paralelo, podendo encontrar todo o seu conteúdo completo no código em C, com os devidos comentários sobre partes especificas da implementação.

A divisão dos dados foi feita em Chunks, ou seja, cada processo do MPI calcula os m vizinhos mais próximos de um subconjunto de partículas:

$$Chunk_{size} = \frac{n}{size} \tag{1}$$

A comunicação é feita utilizando as funções nativas do MPI, como : MPI (init, comm rank, size).

Não há comunicação explícita para troca de partículas entre os processos, o que pode ser um gargalo se n for muito grande, pois todos os processos trabalham com o conjunto completo de partículas.

Assim, a distribuição é feita de maneira estática, o que é um ponto que poderia ser melhorado usando um distribuição de tarefas dinâmicas, como vimos no esquema gerente-trabalhador, em sala. No entanto, como está exposto na seção de resultados, a ociosidade de processos não foi muito relevante para o tamanho de partículas testes que foi fornecido na disciplina, por isso a distribuição em Chunks estática foi aplicada.

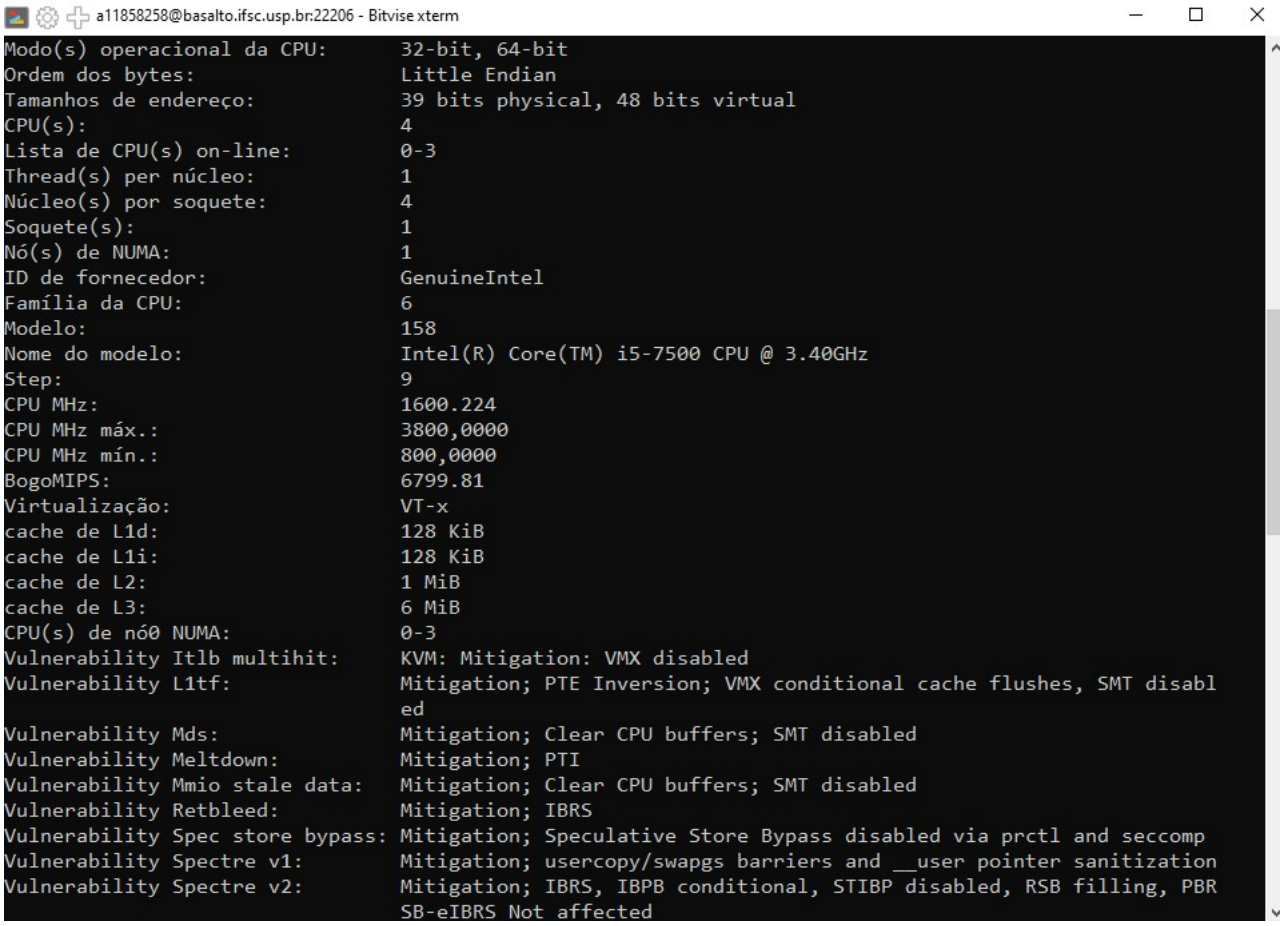
Para a saída, foi aplicada uma redução: A saída de cada processo é consolidada no arquivo final. O processo rank 0 é responsável pela escrita.

A paralelização é de granularidade média: as partículas são divididas uniformemente entre os processos, mas o conjunto completo é necessário para calcular as distâncias. Isso pode levar a overheads de memória e comunicação em máquinas com muitos processos.

Por fim, como veremos nos resultados, principalmente para n muito pequeno, o uso de paralelismo ainda não foi eficiente, muito menos quando comparado ao uso de estruturas como Árvores - KD no algoritimo sequencial.

3 Comparações de Desempenho

Nesta secção, vamos comparar o desempenho das diferentes versões do programa. Para os cálculos, foi usado o processador abaixo:



Vale notar que na linha de saida, os valores de tempo de leitura, tempo de cálculo e tempo de escrita sempre variam. Isso acontece por diversos fatores, como visto em aula: diferente tempos de latência entre outros hardwares com a CPU, localidades (principalmente temporal para as memórias na escrita e leitura), aquecimento e varição de frequência da CPU, distribuição de uso de outras aplicações e etc.

Assim, inicialmente é feito um "aquecimento" do código, rodando ele algumas vezes, com o intuito de minimizar esses efeitos. Em seguida, é tirado os valores da linha de saída 5 vezes. Disso, é feita a média dessas variáveis e então mostradas na tabela abaixo .

Aqui retomo a comparação do algoritimo sequencial vs usando árvores KD: vamos comparar o desempenho do código $t1_{SoA}$ com o do algoritimo usando Árvores d-dimensionais (no nosso caso d=3).

	$t1_{SoA}$	$t1_{kd-tree}$
Tempo de leitura	0.000601	0.000627
Tempo de cálculo	0.002910	0.000278
Tempo de escrita	0.000450	0.008632

Table 1: Para cada caso, foi gerado um arquivo de 1000 partículas aleatórias e calculado os 5 vizinhos mais próximos.

4 Resultados

Vamos começar analisando como os tempos de leitura, cálculo, escrita, variam com n:

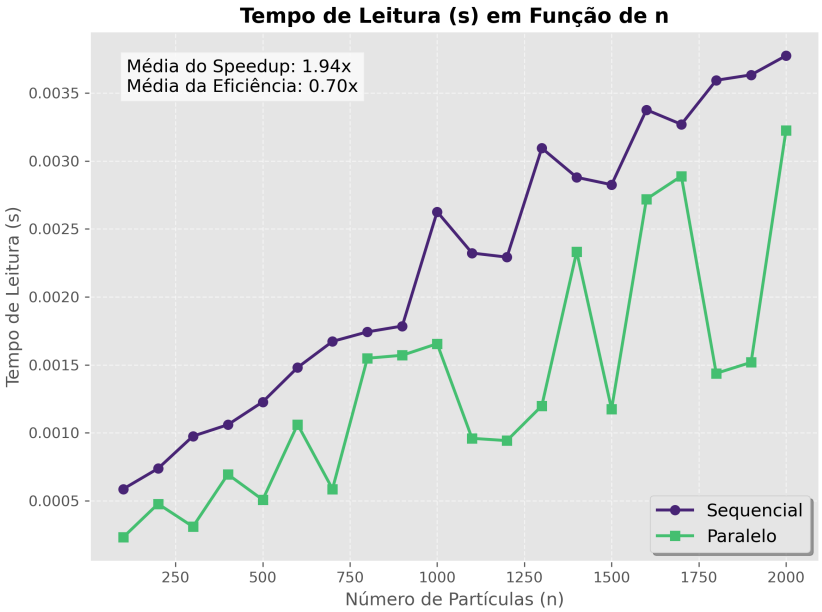


Figure 1: Benchmark usando m= 5 e np - 2

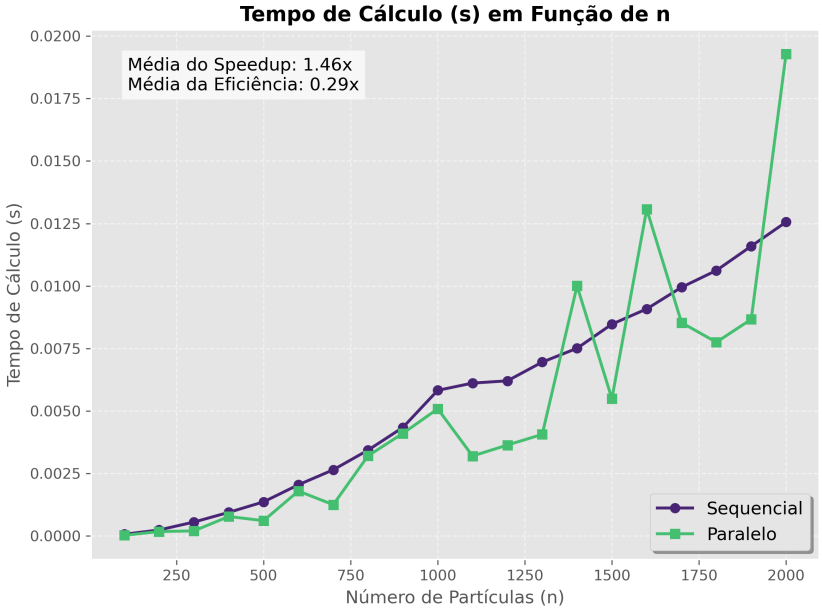


Figure 2: Benchmark usando m= 5 e np - 2

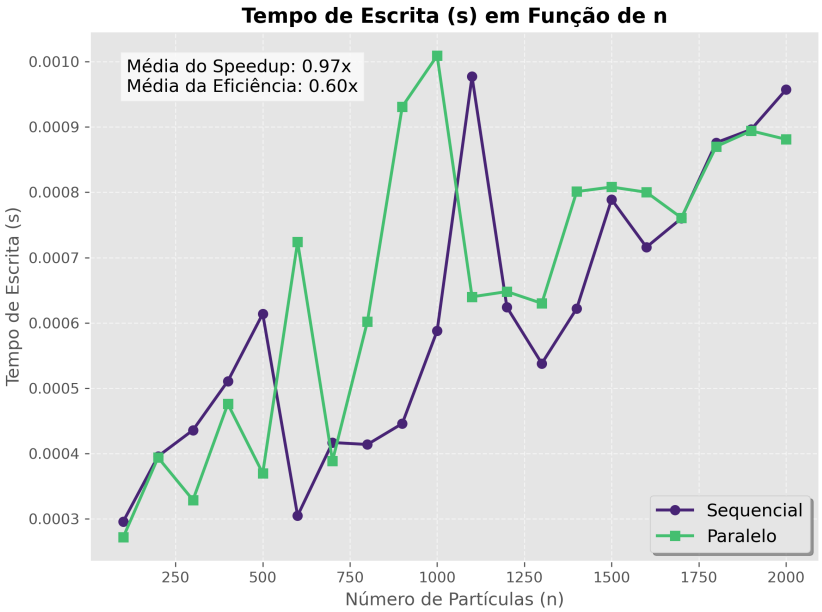


Figure 3: Benchmark usando m= 5 e np - 2

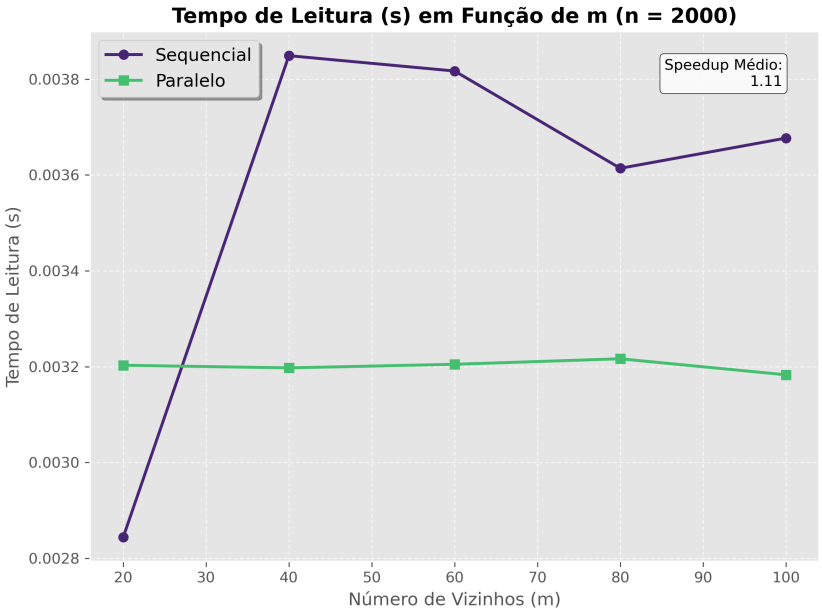


Figure 4: Benchmark usando o arquivo com 2k partículas (large20.pos) e np - 2

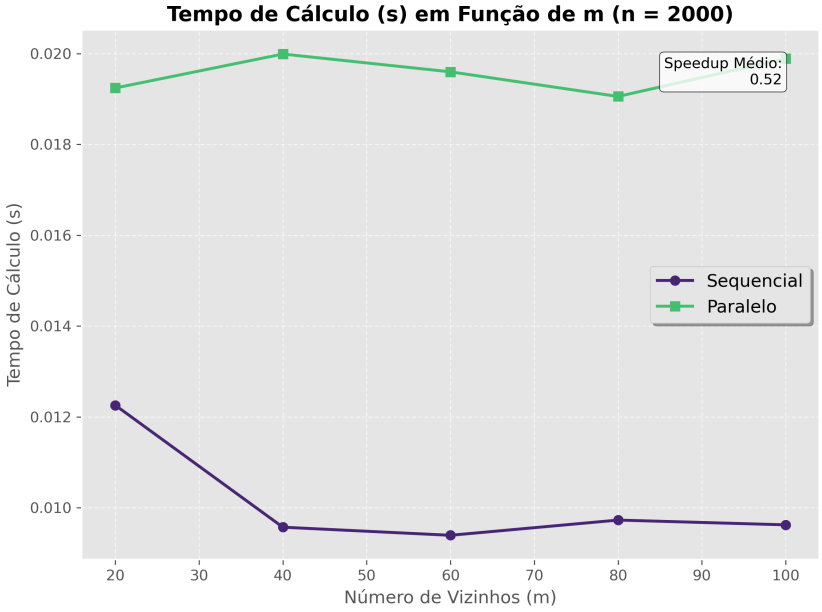


Figure 5: Benchmark usando o arquivo com 2k partículas (large20.pos) e np - 2

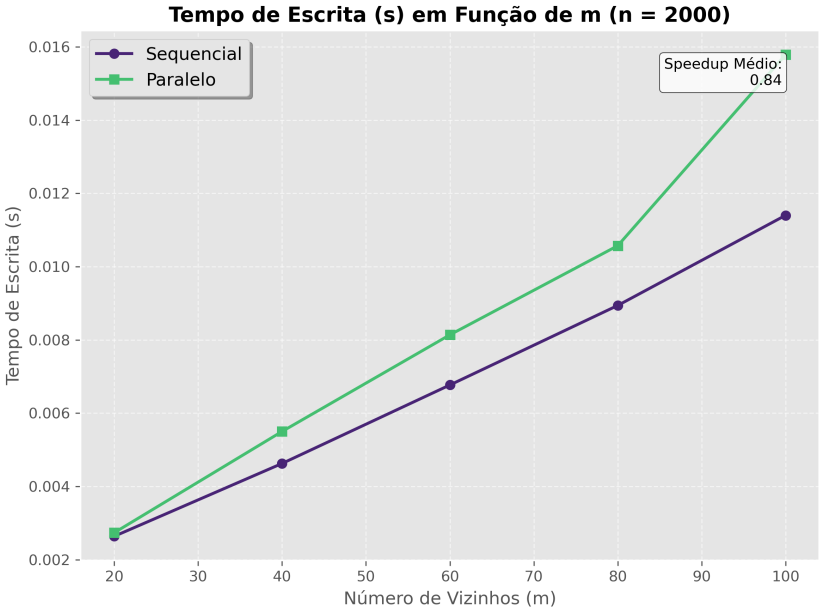


Figure 6: Benchmark usando o arquivo com 2k partículas (large20.pos) e np - 2

Leitura: O tempo de leitura está fortemente associado à latência de I/O e localidade de memória. A leitura sequencial é mais eficiente do que leituras aleatórias, uma vez que as estruturas modernas de memória em disco (principalmente SSDs) favorecem acessos contínuos. Para o paralelismo com MPI, a leitura em um único processo (rank 0) introduz um gargalo natural, já que outros processos dependem dessa etapa inicial para começarem suas computações. O tempo de leitura tende a não ser afetado significativamente pelo número de processos (np), pois não está paralelizado.

Cálculo: O cálculo da distância euclidiana entre pares de partículas possui complexidade teórica $O(n \times m)$, onde n é o número total de partículas e m o número de vizinhos mais próximos analisados.

Em uma abordagem paralela com distribuição estática (chunks), cada processo trabalha em um subconjunto de partículas com o conjunto completo disponível. Apesar do paralelismo, os seguintes aspectos teóricos impactam o tempo de cálculo: Overhead de memória: Cada processo precisa carregar o conjunto completo de partículas, o que pode consumir uma quantidade de memória proporcional a O(n). Isso torna o método suscetível a escalabilidade limitada em arquiteturas com muitas CPUs. Sobrecarga da distribuição estática: A divisão fixa de trabalho pode levar a load imbalance (desequilíbrio de carga) para tamanhos não múltiplos do número de processos ou se houver discrepância nos tempos de processamento de partículas distintas.

Escrita de Dados: O tempo de escrita é influenciado pela redução final dos resultados no processo rank 0. Nesta abordagem, todos os resultados são "gathered" (agregados) em um único processo responsável por consolidar os dados e gerar o arquivo de saída. Essa fase pode introduzir os seguintes gargalos teóricos: O tempo de comunicação entre processos cresce conforme aumentamos o número total de processos np. Para um número elevado, a soma total dos dados e o envio sequencial ao rank 0 pode tornar essa etapa lenta. Além disso, a escrita do arquivo segue limitada pelo hardware de I/O do sistema, fazendo com que essa parte do programa não escale de forma linear com o paralelismo.

Esses resultados corroboram a teoria de que, para casos com n pequeno, a paralelização apresenta ganhos limitados, com a leitura e escrita contribuindo significativamente para o tempo total.

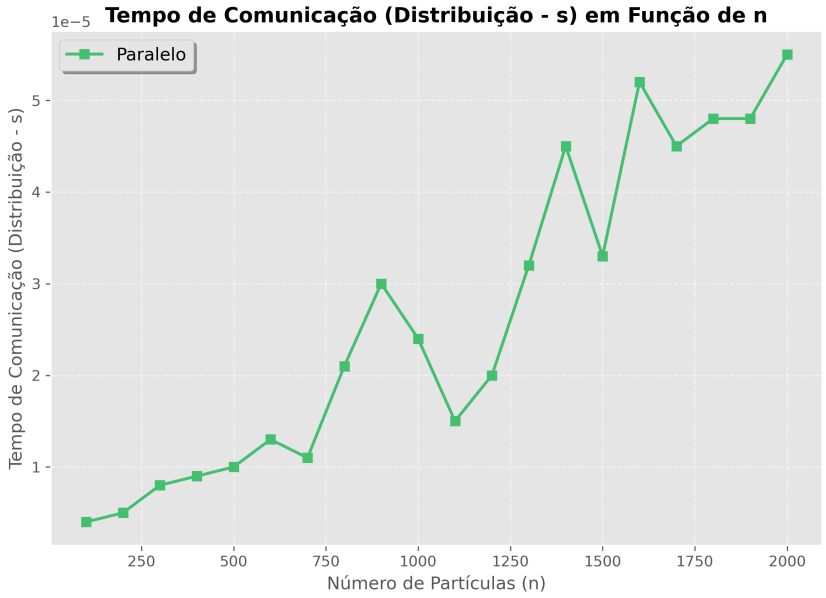


Figure 7: Variação do tempo de gathering dos resultados de dados em paralelo , m = 5 e np = 2

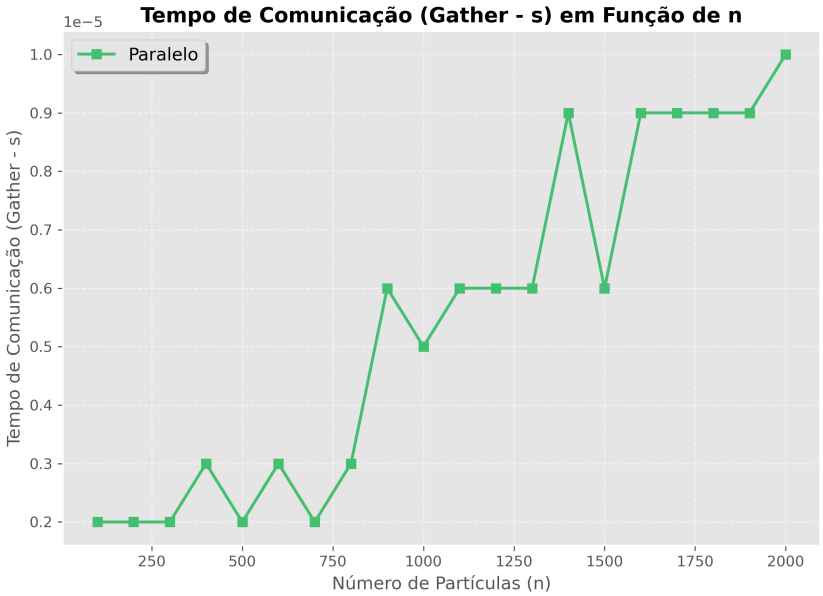


Figure 8: Variação do tempo de gathering dos resultados de dados em paralelo , m = 5 e np = 2

O gathering de dados, ou agregação dos resultados, é uma etapa crucial que impacta o desempenho do programa paralelo quando utilizamos redução estática. O tempo total de gathering segue, em sistemas MPI com agregação sequencial, uma relação aproximada O(np). À medida que aumentamos o número de processos (np), cada um deve enviar seu subconjunto de resultados ao processo responsável (rank 0). Esse envio provoca: Aumento no custo de comunicação, introduzido pelas funções nativas como Reduce e Gather e em grandes valores de np, o custo da latência da comunicação soma-se ao envio dos dados em banda (volume de dados).

Como dito anteriormente, a distribuição de tarefas paralelas é estática, ou seja, há um desequilíbrio que pode causar tempos de espera em processos mais rápidos até que o rank 0 consolide os dados, prejudicando o desempenho do gathering. A etapa de gathering tende a limitar a escalabilidade do programa, especialmente para valores pequenos de n. Quando np cresce, os ganhos de performance obtidos pelo paralelismo são amortecidos pelo tempo adicional da agregação.

Em resumo, o gathering apresenta overhead inevitável conforme o número de processos cresce. Isso enfatiza a importância de técnicas como distribuição dinâmica de tarefas ou comunicações hierárquicas (como binary tree reduce) para mitigar esse efeito em aplicações mais escaláveis.

Por fim, para avaliarmos como a eficiência e o speedup variam com o número de processos.

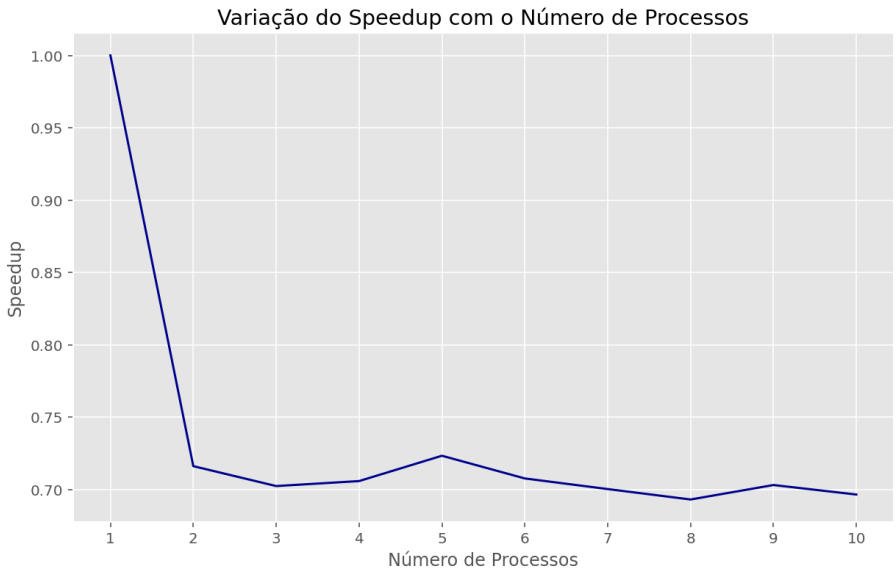


Figure 9: Variação do Speedup, Benchmark usando o arquivo com 2k partículas (large20.pos) e m = 5

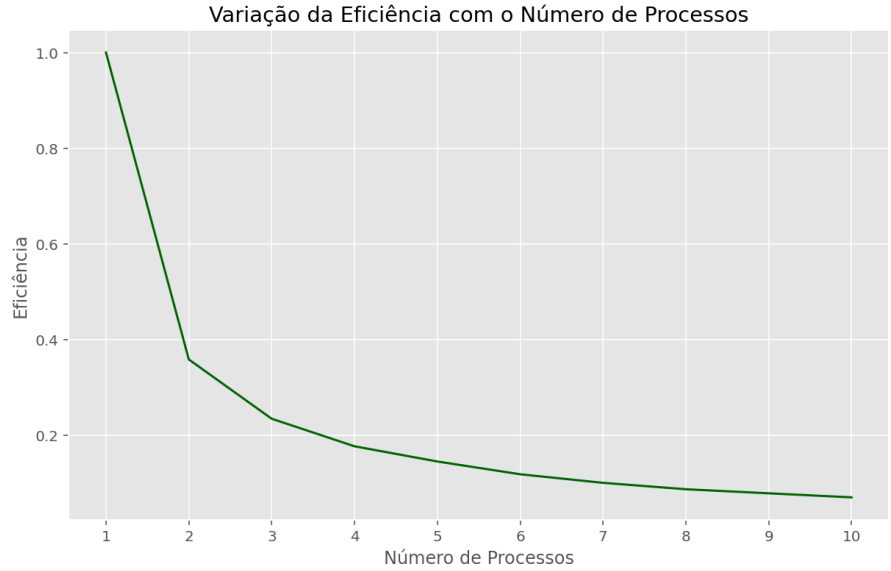


Figure 10: Variação da Efficiência, Benchmark usando o arquivo com 2k partículas (large20.pos) e m = 5

Nesta seção, vamos utilizar duas leis fundamentais da computação paralela: Lei de Amdahl e Lei de Gustafson, para explicar os resultados observados. A Lei de Amdahl define o limite teórico do speedup em um programa paralelo:

$$S = \frac{1}{f_s + \frac{f_p}{np}} \tag{2}$$

Onde f_s é a fração sequencial do código (leitura, escrita e gathering, neste caso) e f_p é a fração paralelizável do cálculo.

Como visto nos resultados, o speedup aumenta inicialmente conforme adicionamos mais processos, mas começa a "saturar" em np = 5. Isso ocorre porque a fração sequencial do programa f_s se torna dominante,

limitando os ganhos do paralelismo. A parte sequencial engloba: leitura inicial do arquivo e escrita final e gathering.

Lei de Gustafson: Caso o problema fosse dimensionado para grandes n, a Lei de Gustafson mostraria que o speedup poderia crescer proporcionalmente ao número de processos. Contudo, com o tamanho fixo dos dados em n=2k, o programa saturou devido à limitação da divisão estática.

A eficiência é definida como:

$$Eff = \frac{S}{np} \tag{3}$$

A eficiência observada diminui conforme np cresce, pois o overhead de comunicação e o tempo ocioso dos processos aumentam. Isso é especialmente perceptível devido à estratégia de granularidade estática, que não permite balanceamento dinâmico. A eficiência decrescente indica a necessidade de estratégias mais avançadas, como distribuição dinâmica ou uso de árvores de redução hierárquica

Por fim, vemos que mesmo usando a paralelização, o caminho usando estruturas como KD-Tress no sequencial, ainda foi bem mais otimizado em tempo, ressaltando o que foi destacado no começo da disciplina: paralelismo nem sempre será eficaz. É sempre melhor, parar e tentar otimizar ao máximo um código sequencial primeiro, antes de sair paralelizando.