Relatório - Trabalho de Paralelização

Alunos: Lucas Ivanov Costa, Vinicius Viana Vieira.
Professor: Guilherme Galante.

Repositório do trabalho: Paralelo-e-Nuvem

De início foi recebido um conjunto de códigos seriais, de onde deveriam ser selecionados apenas três e então paralelizados utilizando dois métodos de paralelização distintos. Os métodos de paralelização vieram de um conjunto de três métodos no qual deveríamos escolher apenas dois.
Os algoritmos escolhidos para a paralelização foram:
- BucketSort;
- Friendly;
- Nbody.

Os métodos de paralelização foram:
- OpenMP;
- OpenMPI.

BucketSort

No arquivo fornecido pelo professor a ordenação ocorre da seguinte forma serial:
- Os baldes são criados inicializados e os valores distribuídos para cada Bucket.
- A ordenação ocorre em cada Bucket um em seguida do outro.
- Usando o método do Insertion Sort.

Para deixar que a ordenação ocorresse de forma paralelizada, foi utilizado a biblioteca OpenMP, que disponibiliza alguns pragmas para serem utilizados. Com isso foi feita da seguinte maneira a paralelização.

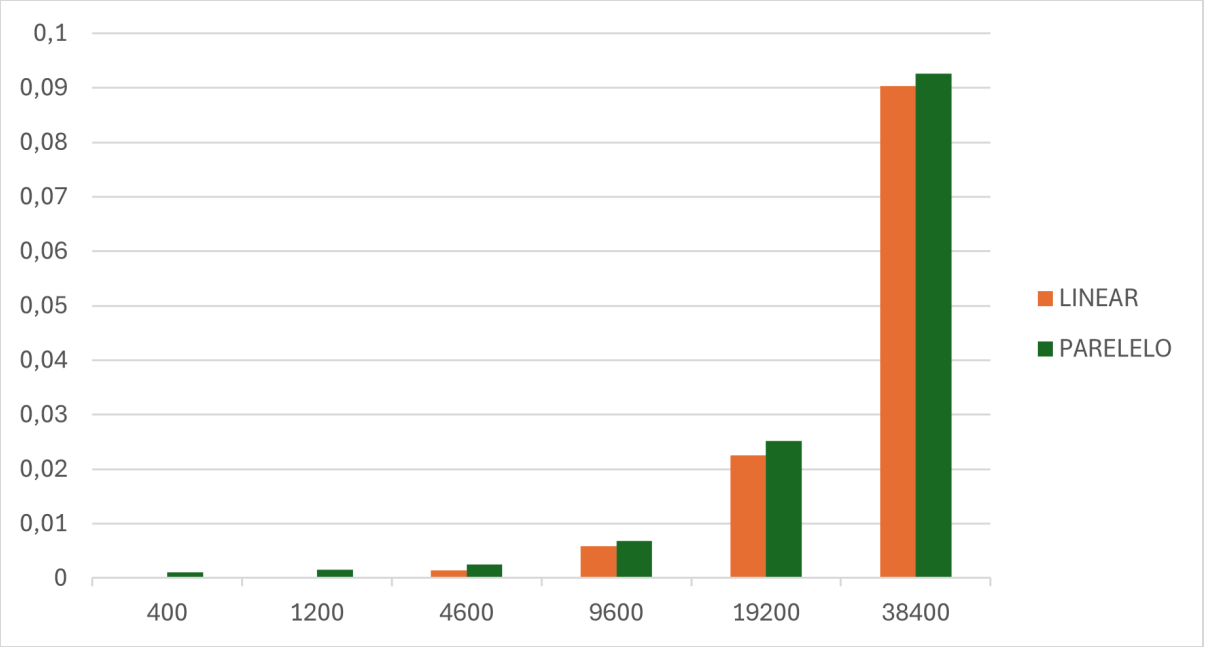
- Com a inserção da biblioteca OpenMP, foi apenas inserido antes do looping for que realizava a chamada do Insertion Sort para cada Bucket um directiva #pragma omp parallel for.

Essa diretiva indica ao processador, que cada iteração do looping deve ser executada em um threading separada.

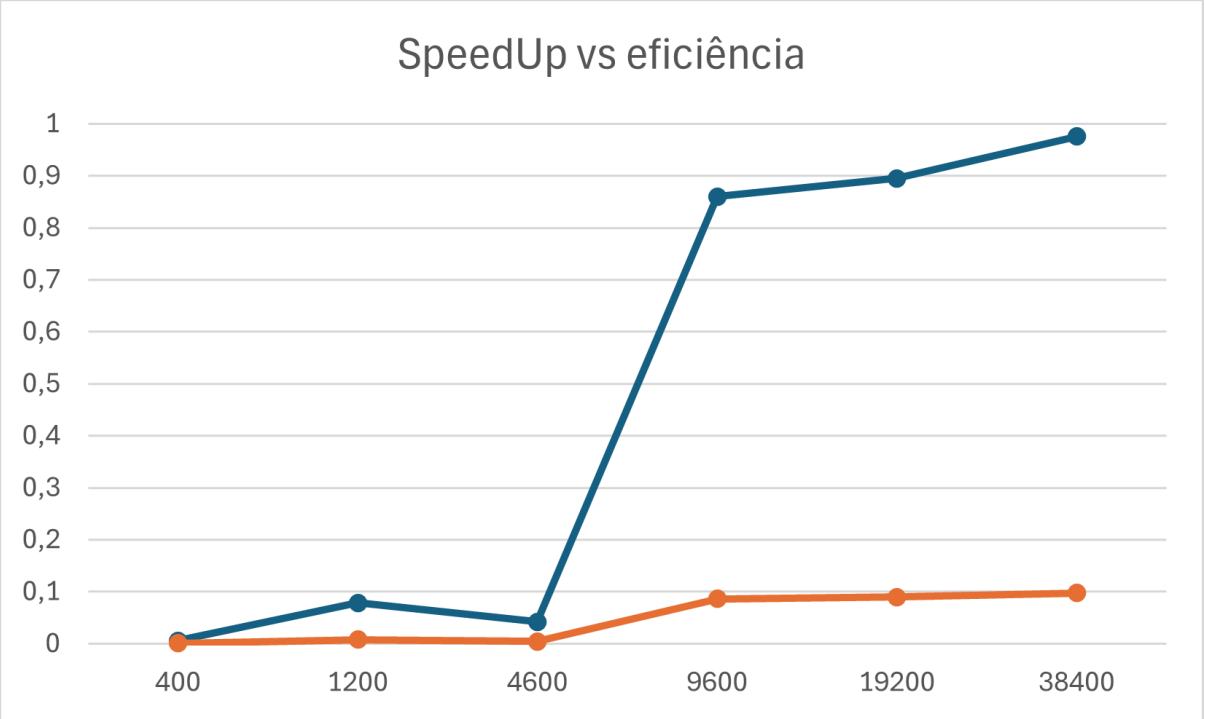
Dados

Em seguida será mostrado alguns dados e métricas referentes ao código paralelizado.

- No seguinte gráfico vemos o tempo gasto em segundos para a execução do código, tanto linear quanto paralelo.



- Aqui podemos ver que para as quantidade de dados passadas para o programa a diferença do tempo de execução para os dois modos foi parecido, mas é visível que quanto maior é o tamanho da entrada, mais vantagem temos com a paralelização.



- Azul: Speedup
- Laranja: Efficiency

Entradas maiores que 50000 travavam o terminal, então para manter uma certa escala, foram feitos apenas os testes com os tamanhos mostrados nos gráficos.

Friendly

No arquivo fornecido pelo professor o processor de encontrar os friendly numbers ocorre da seguinte forma serial:

- É passado como parâmetro dois números que definem um intervalo.
- Para cada número no intervalo fornecido, é calculado a sua razão e a armazena em um vetor.
- Em seguida esse vetor é varrido por dois loopings aninhados, onde é verificado se a razão de dois números são iguais, e se forem, então eles são friendly numbers.

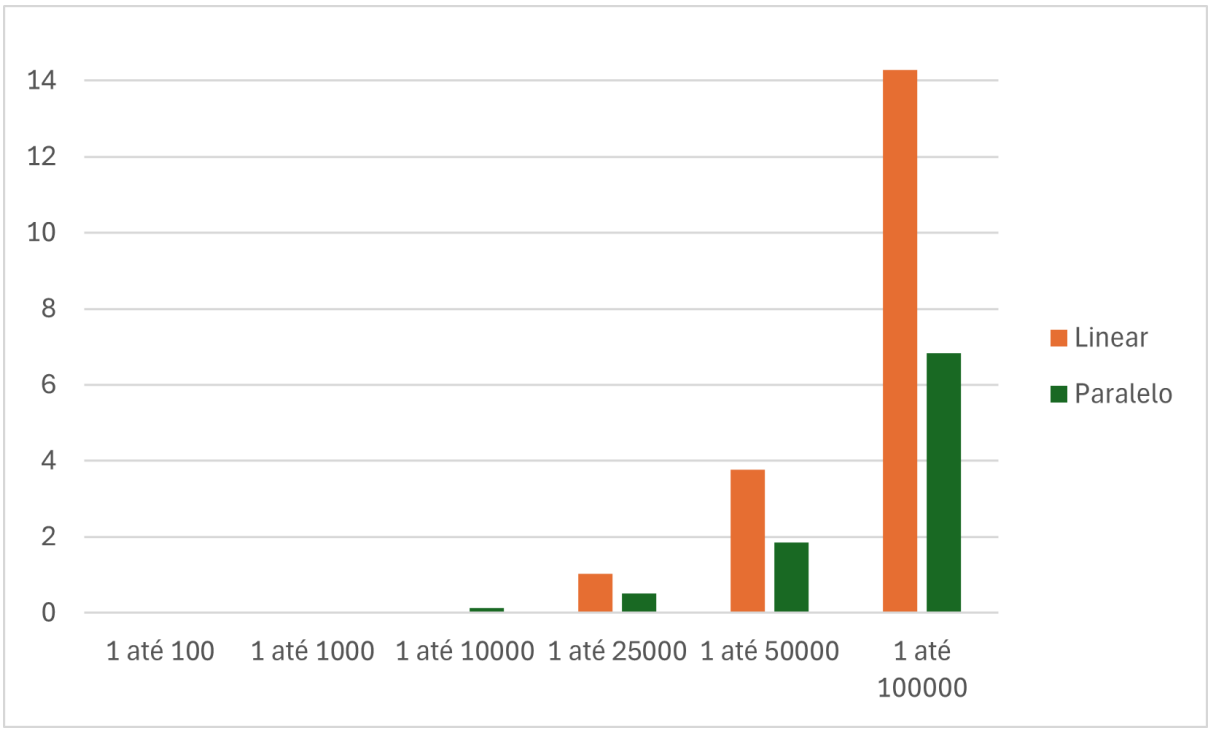
Para a paralelização deste código também foi utilizado a biblioteca OpenMP, e a paralelização foi feita da seguinte forma:

- Com a inserção da biblioteca OpenMP, foi apenas inserido antes do looping for que realizava o cálculo das razões para cada número do intervalo a mesma diretiva que foi utilizada no BucketSort, porém com algumas adições #pragma omp parallel for private(i) schedule(dynamic).
- O schedule(dynamic) foi utilizado para que as iterações fossem distribuídas de forma dinâmica entre os threads, isso foi feito pois o tempo de cálculo da razão de cada número varia muito, e com isso o balanceamento do trabalho entre os threads fica melhor.
- O private(i) foi utilizado para que cada thread tivesse sua própria cópia da variável de controle do looping, evitando assim problemas de concorrência.
- Por último no looping aninhado que verifica as razões iguais, foi adicionado a diretiva #pragma omp for, para que esse looping também fosse paralelizado.

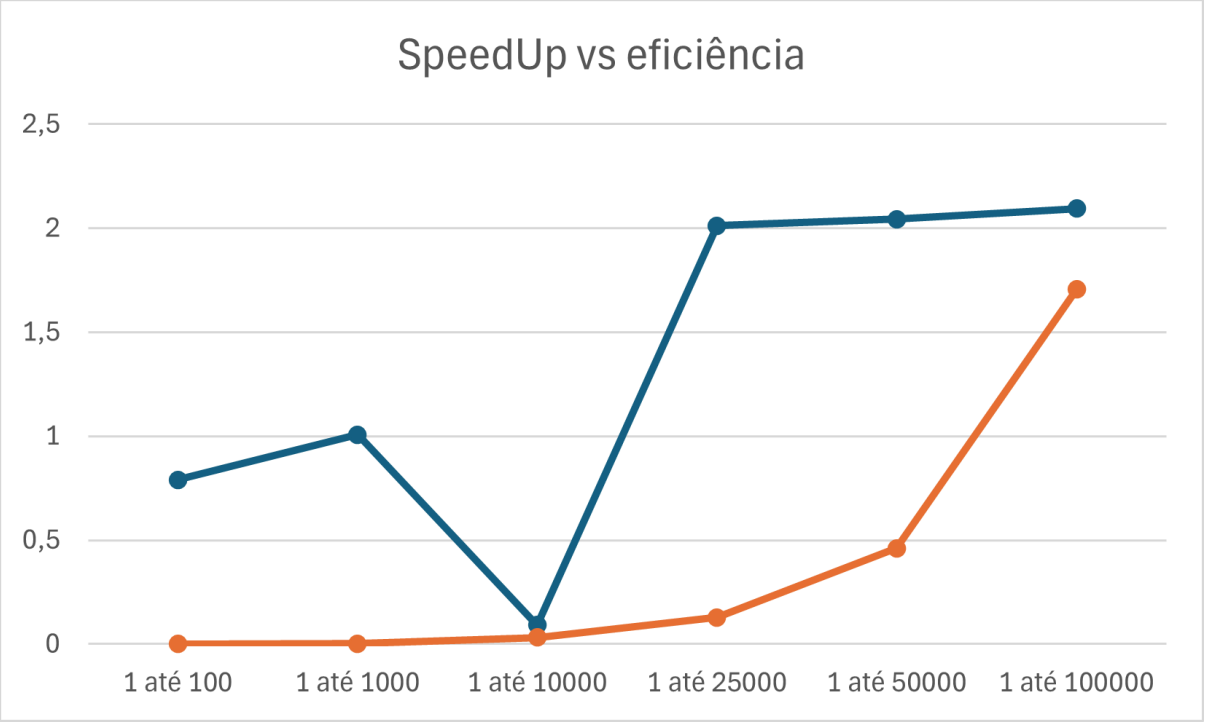
Dados

Em seguida será mostrado alguns dados e métricas referentes ao código paralelizado.

- No seguinte gráfico vemos o tempo gasto em segundos para a execução do código, tanto linear quanto paralelo. Neste código já percebemos que o ganho de tempo com a paralelização é maior.



- Aqui vemos a comparação entre Speedup e eficiência para o código paralelizado, podemos perceber que no terceiro teste realizado houve uma queda no Speedup assim como também ocorreu para o BucketSort, porém a eficiência se manteve mais estável.



- Azul: Speedup
- Laranja: Efficiency

E pelo que é visto, a tendência é de que quanto maior o intervalo, maior seja o ganho com a paralelização.

Nbody

No código fornecido, o simulador de iteração entre partículas ocorre da seguinte forma serial:

- É lido o número de partículas e a quantidade de iterações.
- A cada iteração, calcula a força de cada partícula em relação a todas as outras do sistema (complexidade O(n^2)).
- Com as forças obtidas, as posições são atualizadas e o tempo (dt) é recalculado.

Para a paralelização deste código, foi utilizada a biblioteca MPI, e a lógica foi implementada da seguinte forma:

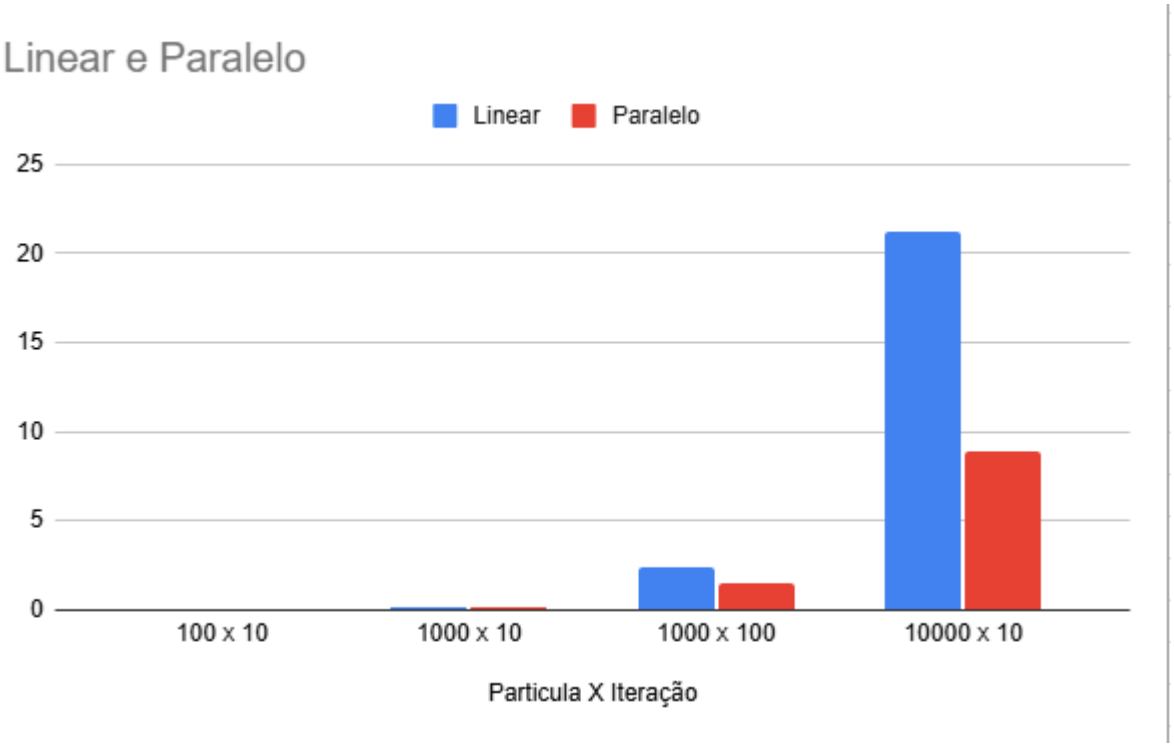
- Distribuição de Carga: O total de partículas (npart) é dividido pelo número de processos (size). Cada processo fica responsável por calcular as forças de um bloco específico de partículas (chunk).
- Comunicação Coletiva:
    - Foi utilizado o MPI_Bcast para enviar os dados iniciais do processo mestre (rank 0) para todos os outros processos.

- Foi utilizado o MPI_Allreduce com a operação MPI_MAX para encontrar a força máxima global. Isso é necessário para que todos os processos ajustem o passo de tempo (dt) de forma sincronizada.
- Independência de Memória: Como o MPI trabalha com memória distribuída, cada processo calcula apenas a sua parte das forças, reduzindo o tempo de processamento proporcionalmente ao número de núcleos utilizados.

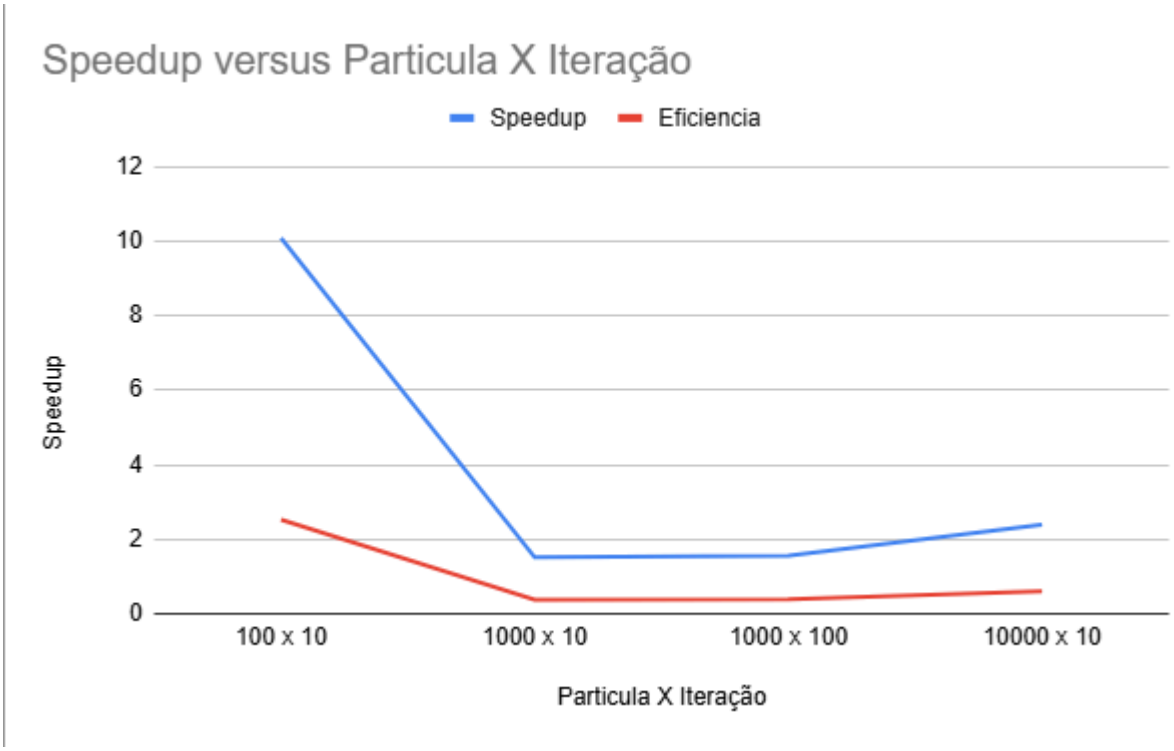
Dados

Em seguida será mostrado alguns dados e métricas referentes ao código paralelizado.

- No seguinte gráfico vemos o tempo gasto em segundos para a execução do código, tanto linear quanto paralelo. Neste código já percebemos que o ganho de tempo com a paralelização é maior.



- Aqui vemos a comparação do Speedup para o código paralelizado. Podemos verificar uma aceleração desde o começo (como eram dois valores pequenos, acabou por dar mais de 10x de aceleração na primeira) até o final, onde vemos uma aceleração de mais de 2x.



- Azul: Speedup
- Vermelho: Eficiência

A tendência é de que quanto maior a quantidade de partículas x iteração, maior será o ganho com a paralelização.