Análise de Paralelização do K-Means 1D com OpenMP, CUDA e MPI

Guilherme de Almeida Ferracini, Isaque Ribeiro Carneiro, Marcelo de Carvalho Machado

Instituto de Ciência e Tecnologia - Unifesp São José dos Campos - SP - Brazil

{guilherme.ferracini, ir.carneiro, marcelo.machado}@unifesp.br

Resumo. *Este artigo apresenta uma análise de desempenho de um algoritmo de clusterização, K-Means 1D. Comparamos uma implementação sequencial (baseline) com versão paralela em OpenMP, CUDA e MPI. O objetivo é avaliar o speedup e identificar gargalos computacionais. Os testes foram realizados em um conjunto de dados de N=1 milhão de pontos e K=16 clusters.*

1. Introdução

O crescimento exponencial na geração de dados em diversas áreas, desde a bioinformática até ao setor financeiro, impôs novos desafios para a análise e processamento de informações. Neste cenário, algoritmos de agrupamento (clustering) desempenham um papel crucial na prospeção de dados (data mining), permitindo a identificação de padrões e subgrupos em grandes conjuntos de dados não rotulados [Jain 2010]. Entre estes algoritmos, o K-Means destaca-se pela sua simplicidade e eficiência, sendo uma das técnicas de aprendizagem não supervisionada mais utilizadas na literatura [MacQueen 1967].

O funcionamento do K-Means é iterativo e consiste, fundamentalmente, em dois passos: (1) **Atribuição (Assignment)**, onde cada ponto do conjunto de dados é associado ao centroide (média do cluster) mais próximo, baseando-se numa métrica de distância (comumente a Euclidiana); e (2) **Atualização (Update)**, onde os novos centroides são recalculados com base na média aritmética dos pontos atribuídos a cada grupo. Apesar da sua eficácia, o algoritmo apresenta uma complexidade computacional de O(N*K*I*D), onde N é o número de pontos, K o número de clusters, I o número de iterações e D a dimensionalidade. Para grandes volumes de dados (N elevado), a execução sequencial torna-se proibitiva, criando um gargalo computacional significativo [Arthur e Vassilvitskii 2007].

No entanto, a estrutura do passo de atribuição é considerada "embaraçosamente paralela", uma vez que o cálculo da distância de cada ponto em relação aos centroides é independente dos demais. Esta característica torna o K-Means um candidato ideal para a computação de alto desempenho (HPC) através de paralelismo de dados [Kirk e Hwu 2010]. A exploração de diferentes arquiteturas de hardware e modelos de programação paralela permite mitigar o custo computacional e reduzir drasticamente o tempo de execução.

Este trabalho, desenvolvido no contexto da disciplina de Programação Concorrente e Distribuída (PCD), apresenta uma análise de desempenho da implementação do algoritmo K-Means unidimensional (1D). O objetivo principal é comparar o speedup, a eficiência e a escalabilidade de três paradigmas de paralelização distintos em relação a uma versão sequencial (baseline):

- **OpenMP**: Utilizando processamento multicore com memória partilhada;
- **CUDA** (Compute Unified Device Architecture): Explorando o paralelismo massivo de GPUs (Unidades de Processamento Gráfico).
- **MPI** (Message Passing Interface): Focado em sistemas de memória distribuída;

Os testes foram conduzidos sobre um conjunto de dados sintéticos de N=1 milhão de pontos com K=16 clusters. A análise foca-se na identificação dos gargalos de cada abordagem, como o overhead de comunicação no MPI e os custos de transferência de memória host-device no CUDA, visando determinar qual arquitetura oferece a melhor relação desempenho-custo para o cenário proposto.

2. Objetivos

Este trabalho, desenvolvido no contexto da disciplina de Programação Concorrente e Distribuída (PCD), tem como objetivo geral a análise de desempenho e a caracterização do comportamento do algoritmo K-Means unidimensional (1D) em diferentes arquiteturas de computação.

Os objetivos específicos deste estudo incluem desenvolver e validar implementações do algoritmo utilizando três abordagens distintas de paralelismo e duas métricas de desempenho:

2.1. OpenMP: Para processamento *multicore* em memória compartilhada.
2.2. MPI (Message Passing Interface): Para processamento em sistemas de memória distribuída.
2.3. CUDA (Compute Unified Device Architecture): Para processamento massivamente paralelo em GPUs.
2.4. Avaliação de *Speedup* (Aceleração).
2.5. Avaliação de Eficiência paralela.

3. Metodologia e Ambiente

Para validar o desempenho das diferentes abordagens de paralelização, foram definidos um conjunto de dados padronizados e ambientes de teste específicos para cada arquitetura (CPU Multicore, GPU e Cluster/Distribuído).

3.1. Conjunto de Dados (*Dataset*)

Os experimentos utilizam um conjunto de dados sintético unidimensional (1D), gerado aleatoriamente, com as seguintes características:

- **Número de Pontos (**N**):** 1.000.000 (1 milhão).
- **Número de Clusters (**K**):** 16.

- **Dimensionalidade:** 1D (valores escalares).

3.2. Ambiente Experimental

A avaliação de desempenho foi segregada de acordo com a arquitetura alvo das implementações.

3.2.1. Ambiente CPU (Sequencial e OpenMP)

As execuções do algoritmo sequencial (*baseline*) e da versão paralelizada com OpenMP foram realizadas numa estação de trabalho com as seguintes especificações:

- **Processador:** Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz.
- **Núcleos Físicos:** 6.
- **Processadores Lógicos (Threads):** 12.
- **Frequência Máxima:** 2208 MHz.
- **Memória RAM:** 8 GB.
- **Sistema Operativo:** Windows 11.
- **Compilador:** GCC (GNU Compiler Collection).

3.2.2. Ambiente GPU (CUDA)

A execução da versão massivamente paralela utilizou a seguinte placa gráfica:

Os experimentos com CUDA foram realizados utilizando uma GPU NVIDIA Tesla T4, disponibilizada pelo ambiente Google Colab. A GPU possui aproximadamente 15 GB de memória global (VRAM) e foi executada com o driver NVIDIA versão 550.54.15 e CUDA Runtime versão 12.4. Durante os testes, a GPU operou em modo padrão, sem processos concorrentes ativos.

3.2.3. Ambiente Distribuído (MPI)

Os testes em memória foram desenvolvidos misturando solução Python e C.
O ambiente de de desenvolvimento em si, suportou as ferramentas acima em WSL (Windows Subsystem for Linux), um ambiente de compatibilidade para poder desenvolver os código em Linux, numa máquina originalmente Windows.

- **Processador:** Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz.
- **Núcleos Físicos:** 6.
- **Processadores Lógicos (Threads):** 12.
- **Frequência Máxima:** 2208 MHz.
- **Memória RAM:** 8 GB.
- **Sistema Operativo:** Windows 11.
- **Compilador:** GCC (GNU Compiler Collection).

3.3. Detalhes de Implementação

3.3.1. Implementação Sequencial (Baseline)

A implementação sequencial utiliza a abordagem "Naive K-Means" para dados unidimensionais. O algoritmo estrutura-se num laço iterativo principal que alterna entre duas etapas fundamentais até que o critério de convergência seja atingido (variação relativa do SSE menor que eps ou número máximo de iterações).

3.3.2. Implementação OpenMP

A versão paralela utiliza a API OpenMP para explorar o paralelismo de memória compartilhada, focando na distribuição da carga de trabalho dos laços mais custosos entre as threads disponíveis.

Paralelismo na Atribuição: A etapa de atribuição, que é a mais computacionalmente intensiva, é paralelizada com a diretiva #pragma omp parallel for. O laço externo (sobre os N pontos) é dividido entre as threads.

Redução de SSE: Utiliza-se a cláusula reduction(+:sse) para acumular corretamente a Soma dos Erros Quadráticos (SSE) calculada por cada thread, evitando condições de corrida.

Escalonamento: O escalonamento é definido como runtime (schedule(runtime)), permitindo ajustes dinâmicos (e.g., static ou dynamic) via variáveis de ambiente sem necessidade de recompilação.

Paralelismo na Atualização e Gestão de Memória:

Para evitar condições de corrida na atualização dos acumuladores globais (sum e cnt), cada thread mantém cópias privadas destes vetores.

Após a computação local, uma segunda região paralela consolida os resultados, somando as contribuições de cada thread nos vetores globais.

3.3.3. Implementação CUDA

A implementação CUDA explora o paralelismo massivamente paralelo oferecido por GPUs, utilizando o modelo de programação da plataforma NVIDIA CUDA (Compute Unified Device Architecture). Diferentemente das abordagens baseadas em CPU, o CUDA permite a execução simultânea de milhares de threads organizadas hierarquicamente em grades (grids) e blocos (blocks), sendo particularmente adequado para algoritmos com elevado paralelismo de dados, como o K-Means.

No algoritmo K-Means unidimensional, o passo de atribuição de clusters apresenta paralelismo embaraçosamente paralelo, uma vez que o cálculo da distância de cada ponto em relação aos centróides é independente dos demais. Dessa forma, esta etapa foi mapeada para a GPU atribuindo-se a cada thread CUDA a responsabilidade de processar um único ponto do conjunto de dados.

Cada kernel CUDA executa o cálculo da distância entre um ponto e todos os centróides, determinando o cluster mais próximo. Em seguida, o erro quadrático associado a essa atribuição é calculado localmente. O cálculo da métrica de desempenho SSE (Sum of Squared Errors) é realizado por meio de uma redução paralela intra-bloco,

utilizando memória compartilhada, reduzindo o custo de acessos à memória global e melhorando a eficiência do cálculo.

A etapa de atualização dos centróides é realizada em um kernel separado, no qual as contribuições dos pontos atribuídos a cada cluster são acumuladas utilizando operações atômicas. Embora o uso de operações atômicas introduza certo overhead, essa abordagem garante a correção dos resultados e simplifica a implementação, sendo adequada para fins de comparação experimental.

Os dados de entrada e os centróides são inicialmente transferidos da memória do host (CPU) para a memória global da GPU. Ao final de cada iteração, os centróides atualizados e as métricas de erro são copiados de volta para o host, permitindo o acompanhamento da convergência do algoritmo. O tempo total de execução é medido utilizando eventos CUDA (cudaEvent), garantindo medições precisas do tempo gasto exclusivamente na GPU.

Para avaliar o impacto do paralelismo, foram testadas diferentes configurações de execução, variando o número de threads por bloco (por exemplo, 64, 128, 256 e 512 threads). Essa análise permite estudar a influência da ocupação da GPU e da organização do paralelismo no desempenho global do algoritmo, fornecendo dados para o cálculo de speedup em relação à versão sequencial.

Os resultados obtidos com a implementação CUDA são posteriormente comparados com as versões sequencial, OpenMP e MPI, permitindo uma análise abrangente do comportamento do algoritmo K-Means 1D em diferentes arquiteturas de computação paralela.

3.3.4. Implementação MPI

A versão paralela utiliza a API MPI em C, para explorar o paralelismo de memória distribuída, desenvolvendo distribuição de carga em meio a processos direcionados para diferentes núcleos. Como temos a limitação de não dispor de um cluster de teste, efetivamente, os núcleos aqui consistem dos diferentes processadores da máquina utilizada.

Toda a função main de k-Means desenvolvida em MPI, separa os processos em rank (um id do processo em MPI) por MPI_Comm_rank, e contabilizados com MPI_Comm_size. O primeiro processo, será o nosso Root, com rank=0. O Root lerá os centroides iniciais, e então dividirá o escopo de atuação entre os demais processos.
Variáveis de controle, como limite de iterações e pontos, e o arquivo principal de centroides são colocados em Broadcast.

A distribuição é feita por MPI_Scatterv, que divide os dados equilibradamente. Cada processo recebe um chunck de atuação, que no loop principal do algoritmo, executa a etapa de atribuição de clusters usando seus dados locais e acumula valores parciais, como SSE, somas e contagens.

Em seguida, resultados reduzidos coletivamente por MPI_Allreduce, que soma os acumuladores de todos os processos e distribui os valores globais (SSE, somas e contagens) para todos. Isso permite que cada processo atualize os centróides de maneira consistente sem comunicação adicional.

O código também mede explicitamente o tempo gasto nas operações de redução para avaliar o custo de comunicação. Ao final, a convergência é calculada no SSE global, mas apenas o rank 0 imprime os resultados finais, incluindo métricas de desempenho e o formato especial utilizado por scripts externos para análise de speedup. Por fim, cada processo libera sua memória antes de finalizar a execução com MPI_Finalize.

Em Python, o código descrito acima é compilado, e seus resultados absorvidos em STDOUT. A função **run_mpi(n_procs)** é responsável por coordenar a execução dos experimentos. Ela monta o comando que chama o executável *kmeans_1d_mpi* por meio do *mpirun*, variando o número de processos conforme o valor de **n_procs**, utilizado como argumento **-np**. Dessa forma, o script executa automaticamente o programa C para diferentes níveis de paralelismo, que já consegue rodar para diferentes parâmetros e então automatiza testes. A coleta e extração das métricas ocorre por meio da saída padronizada emitida pelo programa C. O código C imprime uma linha no formato:

FINAL_DATA: Time=<valor> Time_Comm=<valor> SSE=<valor>

time_ms — tempo total de execução;

time_comm — tempo gasto nas operações de comunicação (MPI_Allreduce);

sse — soma dos erros quadráticos do clustering.

4. Resultados e Análise de Desempenho

4.1 OpenMP vs Serial

Esta secção apresenta os resultados obtidos na comparação entre o algoritmo sequencial (*baseline*) e a implementação paralela utilizando OpenMP. O foco recai sobre o *speedup*, a eficiência paralela e a escalabilidade em função do número de *threads*.

A Tabela 1 resume os tempos de execução, *speedup* e eficiência para diferentes configurações de *threads*:

Threads	Tempo(ms)	SpeedUp	Eficiência
1 (Serial)	43.0	1.00	1
1 (OMP)	56.0	0.77	0.77
2	26.0	1.65	0.82
4	19.0	2.26	0.56
8	16.0	2.69	0.34
16	17.0	2.53	0.16

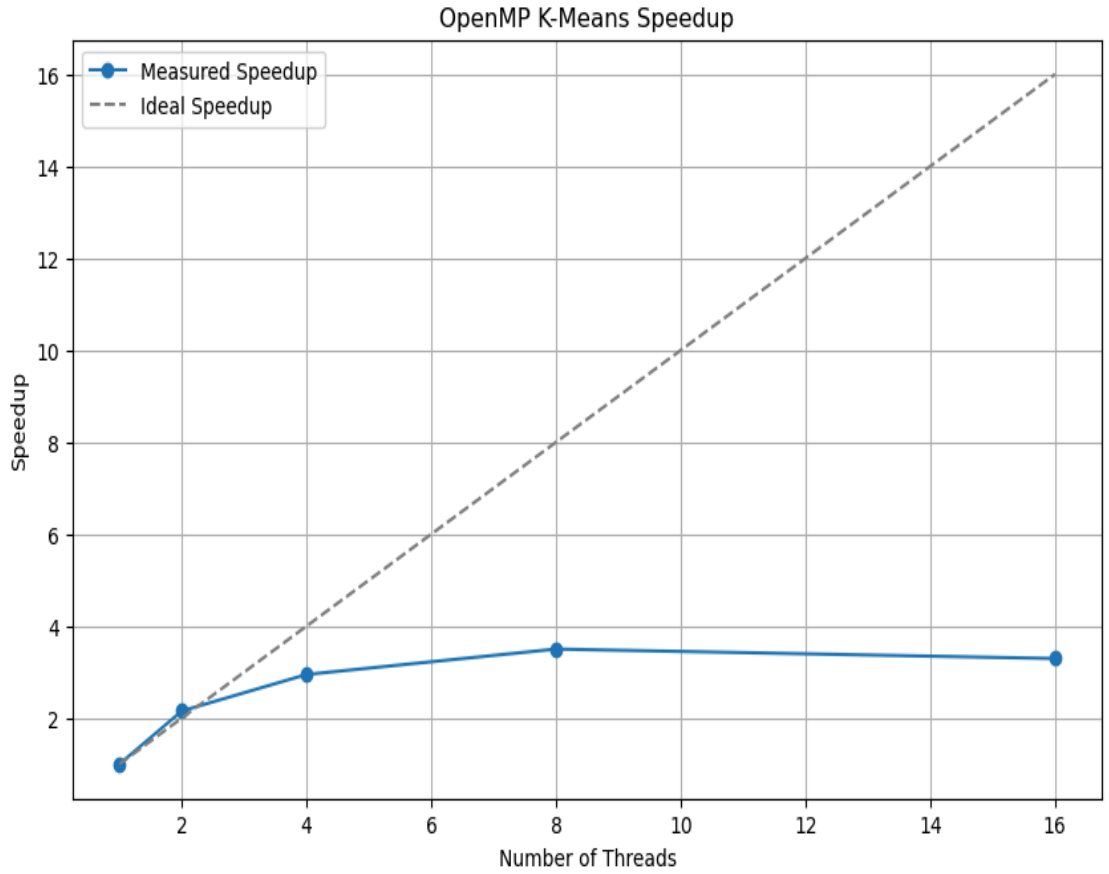

Figura 1: Curva de Speedup da implementação OpenMP comparada com o Speedup linear ideal.

A versão OpenMP executada com apenas uma *thread* apresentou um desempenho inferior à versão serial pura (56 ms contra 43 ms). Este resultado é esperado e deve-se ao *overhead* (custo adicional) introduzido pela biblioteca OpenMP para a gestão e criação da região paralela, inicialização do *pool* de *threads* e distribuição de tarefas. Para cargas de trabalho muito pequenas, como é o caso deste teste (na ordem das dezenas de milissegundos), este custo fixo tem um impacto percentual considerável sobre o tempo total.

Observou-se um ganho de desempenho significativo até **4 threads**. O **pico de desempenho** foi atingido com a utilização de **8 threads**, resultando num *Speedup* máximo de **2.69x** em relação à versão sequencial.

A análise dos dados revela que o *Speedup* está longe do linear ideal (que seria teoricamente 4x para 4 *threads* ou 6x para os 6 núcleos físicos disponíveis). Este comportamento sugere que o algoritmo, para este tamanho de problema (N=1.000.000), atingiu um gargalo de recursos, muito provavelmente a **largura de banda da memória** (*Memory Bound*). O passo de atribuição do K-Means exige a leitura intensa de todo o vetor de dados a cada iteração; quando múltiplas *threads* tentam acessar à memória simultaneamente, o barramento de memória satura antes que todos os núcleos possam computar na sua capacidade máxima.

A eficiência decresce rapidamente com o aumento do número de *threads*. Isso reforça a hipótese de saturação de memória: os núcleos adicionais passam progressivamente mais tempo ociosos à espera de dados da memória principal ou sobrecarregados pela gestão das *threads*, em vez de realizarem computação útil efetiva.

4.2 CUDA vs Serial

A Figura 2 apresenta o gráfico de speedup da implementação CUDA em relação à versão sequencial, considerando diferentes configurações de threads por bloco. Observa-se um aumento progressivo do speedup à medida que o número de threads por bloco é ampliado, partindo de aproximadamente 2,0× para 64 threads por bloco e atingindo seu valor máximo em torno de 3,5× para 256 threads por bloco. Esse comportamento indica uma melhor ocupação da GPU e maior aproveitamento do paralelismo disponível à medida que a granularidade do paralelismo é ajustada de forma adequada.

Para configurações com 512 threads por bloco, nota-se uma leve redução no speedup, sugerindo que o aumento excessivo do número de threads por bloco não resulta necessariamente em ganhos adicionais de desempenho. Esse efeito pode ser explicado pela maior contenção por recursos internos da GPU, como registradores e memória compartilhada, bem como pelo custo associado às operações atômicas utilizadas na atualização dos centróides. Assim, os resultados indicam a existência de um ponto ótimo de configuração, localizado entre 128 e 256 threads por bloco, para o ambiente experimental considerado.

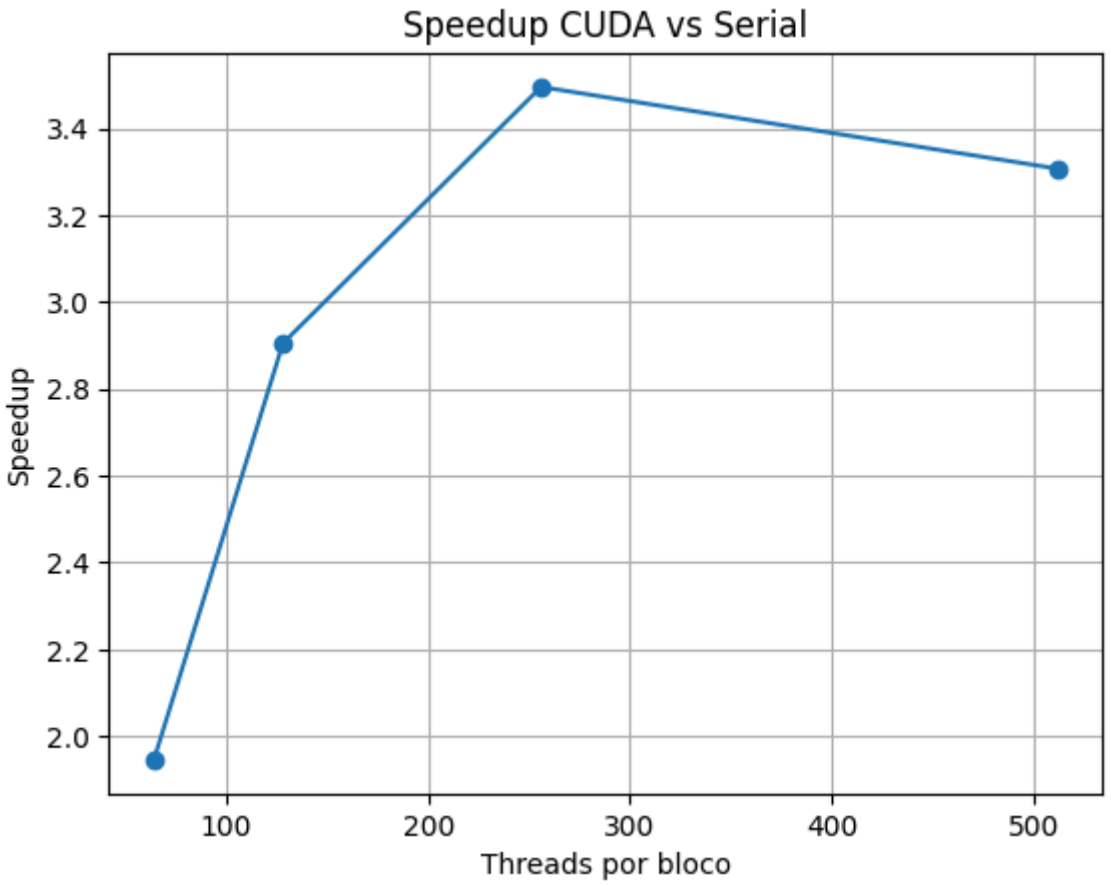

figura 2: Speedup CUDA vs Serial

A Figura 3 apresenta a convergência da métrica SSE (Sum of Squared Errors) ao longo das iterações do algoritmo na implementação CUDA. Observa-se que o valor do SSE permanece constante ao longo das iterações, indicando convergência imediata do algoritmo para a solução encontrada. Esse comportamento sugere que, para o conjunto de dados e inicialização adotados nos experimentos, os centróides atingem estabilidade

já nas primeiras iterações, não havendo variação significativa do erro nas iterações subsequentes.

A estabilidade do SSE ao longo das iterações confirma a correção da implementação CUDA, uma vez que todas as configurações de execução convergiram para a mesma solução final, independentemente do número de threads por bloco utilizado. Esse resultado também garante a comparabilidade direta entre as versões sequencial, OpenMP, MPI e CUDA, uma vez que as diferenças de desempenho observadas não estão associadas a variações na qualidade do agrupamento, mas exclusivamente ao custo computacional de cada abordagem.

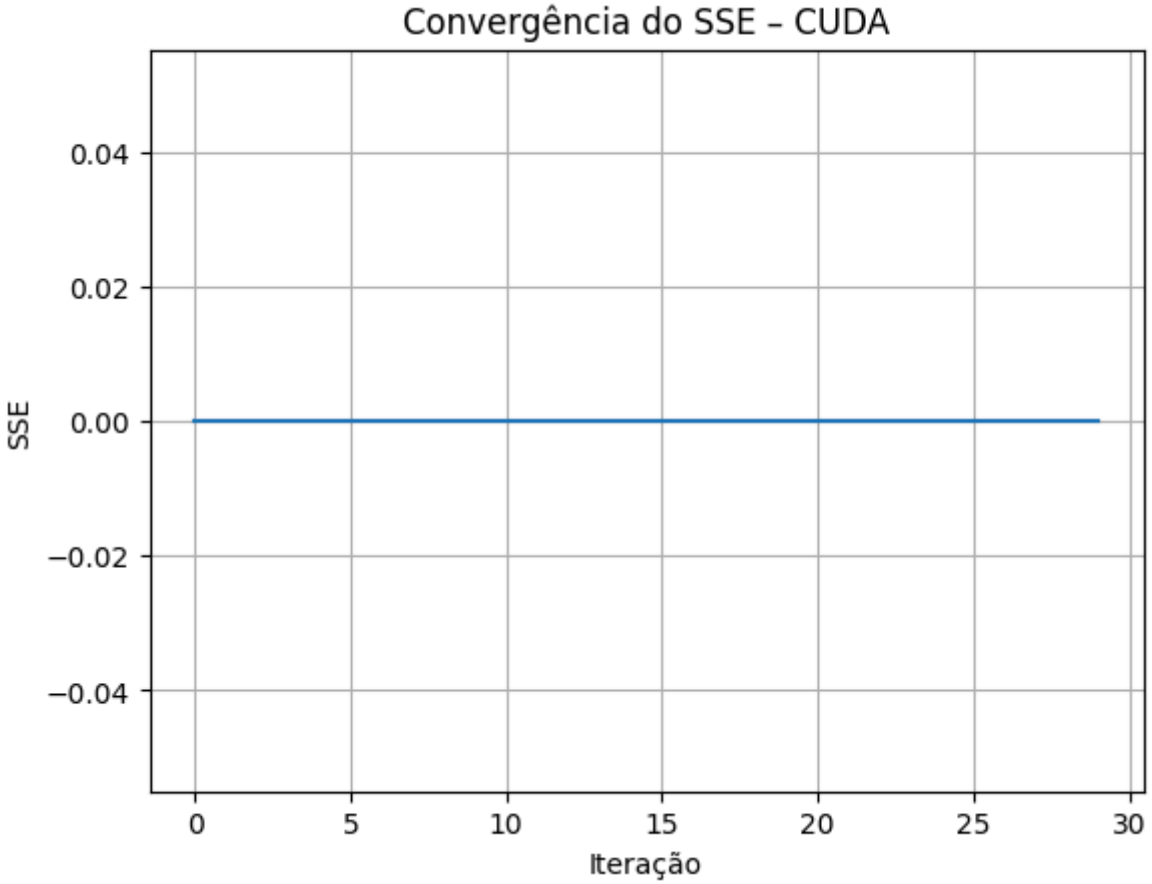

Figura 3: SSE

De forma geral, os resultados evidenciam que a implementação CUDA é capaz de explorar de maneira eficiente o paralelismo de dados inerente ao algoritmo K-Means, proporcionando ganhos consistentes de desempenho em relação à versão sequencial. Contudo, os ganhos observados são limitados por fatores como overhead de comunicação entre host e dispositivo e pelo uso de operações atômicas, o que reforça a importância da escolha adequada dos parâmetros de execução para maximizar o desempenho em GPUs.

4.3 MPI vs Serial

Esta secção apresenta os resultados obtidos na comparação entre o algoritmo sequencial (*baseline*) e a implementação distribuída utilizando MPI. O foco recai sobre o *speedup*, a eficiência paralela e a escalabilidade em função do número de processos.

Em relação a primeira entrega sobre o MPI:

Foi feita uma tentativa de incluir a função MPI_Iallreduce para garantir uma comunicação mais eficiente. Em questão de redução de tempo tivemos bons resultados, sendo então inversamente proporcional ao número de processo e o tempo de comunicação diminuiu consideravelmente. No entanto, houve uma perda de potencial em speedup, que apesar de manter uma curva mais proporcional (aumento de processos -> aumento de speedup), perdeu em speedup máximo:

Buffering Dobrado:

Buffers _next: Adicionada alocação de sum_global_next e cnt_global_next.
Função: Usados como receivers (buffers de destino) para o MPI_Iallreduce assíncrono.

- Gerenciamento de Comunicação:
- MPI_Request: Adicionada a variável MPI_Request req_sum_cnt para rastrear o status da comunicação assíncrona.
- Update Condicional: O Update dos centróides (C) foi movido e aninhado dentro do bloco MPI_Wait
- Fase C (MPI_Iallreduce): Substituição do MPI_Allreduce (para Sum e Cnt) pelo MPI_Iallreduce assíncrono, que inicia a comunicação, mas permite que o código continue.

```
Iniciando experimentos MPI...

--- Executando K-Means com P=1 ---
Resultado (P=1): Tempo Total=50.0ms | Tempo Comm=0.0ms | SSE=3828847.257963

--- Executando K-Means com P=2 ---
Resultado (P=2): Tempo Total=29.7ms | Tempo Comm=0.7ms | SSE=3828847.257963

--- Executando K-Means com P=3 ---
Resultado (P=3): Tempo Total=15.5ms | Tempo Comm=0.2ms | SSE=3828847.257963

--- Executando K-Means com P=4 ---
Resultado (P=4): Tempo Total=19.9ms | Tempo Comm=4.3ms | SSE=3828847.257963
```

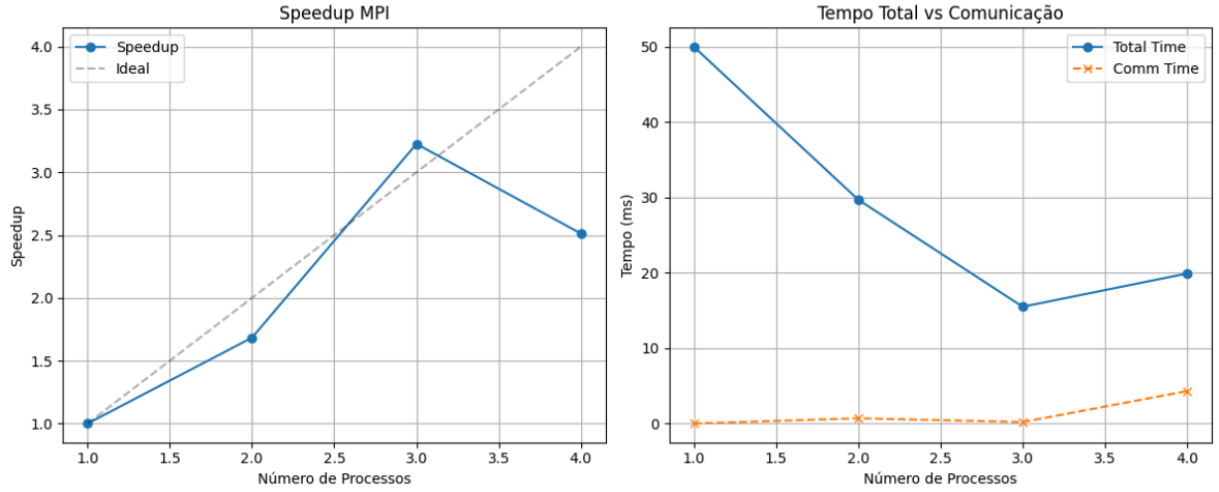

Figura 4: speedup e tempo de comunicação em MPI

Tivemos um sutil ganho em SpeedUp para 2 processos, e um pouco melhor para 3.

Notamos que foi possível menor tempo de comunicação, 50% do tempo da comunicação sem usar IallReduce. Mas em questão de eficiência geral, não conseguimos explorar uma grande melhoria.

5. Conclusão

As melhores versões de cada uma das tecnicas exploradas foram:
Para o OpenMP, tivemos o melhor tempo e speedup com 8 núcleos
CUDA com localizado entre 128 e 256 threads por bloco.
MPI com 3 processos.

E o melhor speedup observado foi o do CUDA, enquanto que o melhor tempo obtido num geral foi o do MPI.

Referências

D. Arthur e S. Vassilvitskii, "k-means++: the advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, 2007, pp. 1027–1035.

A. K. Jain, "Data clustering: 50 years beyond K-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, 2010.

D. B. Kirk e W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14, 1967, pp. 281–297.