Introdução à Programação Paralela e Distribuída (EMA973815)

Relatório do Projeto Final "Clusterização K-Means Paralelo"

Bacharelado em Ciências da Computação - UNESP/RC

Felipe Rovigatti Delfino

Joelle Ramos da Silva

Leticia Cristofoletti Lunardi

Lucas De Souza Silva

Mateus Carrinho Joaquim

Thiago Inouye Miyazaki

Docente:

Alexandro José Baldassin

Rio Claro
2025

Sumário

Resumo

Este trabalho apresenta a paralelização do algoritmo K-Means utilizando três abordagens distintas: Pthreads, OpenMP e OpenMPI. O objetivo é analisar comparativamente o desempenho obtido por cada técnica, observando tempo de execução, speedup, overhead e escalabilidade. Para isso, foram desenvolvidas versões sequenciais e paralelas do algoritmo, aplicadas sobre diferentes volumes de dados a fim de avaliar o impacto das estratégias de paralelização.

Os resultados mostram que cada modelo apresenta vantagens específicas de acordo com a arquitetura de execução e com a intensidade de comunicação e sincronização envolvidas. Enquanto Pthreads e OpenMP demonstram excelente desempenho em ambientes compartilhados, OpenMPI se destaca em cenários distribuídos, embora apresente maior custo de comunicação. A análise evidencia que a escolha da estratégia ideal depende diretamente das características do problema e da infraestrutura disponível.

Palavras-chave: Paralelização, OpenMP, Pthreads, OpenMPI, K-Means.

Introdução

Este projeto foi desenvolvido como trabalho final da disciplina Introdução à Programação Paralela e Distribuída e tem como objetivo explorar, implementar e comparar diferentes paradigmas de paralelização aplicados ao algoritmo K-Means. O trabalho busca analisar de que forma os modelos de programação paralela, como Pthreads, OpenMP e OpenMPI, impactam o desempenho de uma aplicação real, considerando fatores essenciais como estratégias de divisão de trabalho, custos de sincronização, comunicação entre threads e processos, overhead introduzido e capacidade de escalabilidade.

O estudo inclui, adicionalmente, uma avaliação experimental sistemática que compara a versão sequencial do algoritmo com suas respectivas versões paralelizadas. Nessa análise, são avaliados o tempo de execução, o speedup obtido, o impacto da comunicação, a eficiência no uso dos recursos computacionais e as limitações observadas em cada abordagem adotada.

Dessa forma, o projeto não se limita a mensurar melhorias de desempenho, mas também busca compreender os fatores que justificam as diferenças entre os paradigmas de paralelização, contribuindo para o estabelecimento de critérios que auxiliem na escolha da estratégia mais adequada para distintos contextos de execução e infraestruturas computacionais.

Estratégias de Paralelização

Pthreads

Fase de Atribuição

Na fase de atribuição, associamos cada ponto ao cluster mais próximo. Isso é feito pela função assign_points_to_cluster, que percorre aproximadamente M / número_de_threads pontos para cada thread. A divisão é determinada por um intervalo [inicio, fim), onde "inicio" marca o começo do bloco (chunk) que cada thread deve processar.

Durante a iteração, a thread calcula a distância Euclidiana ao quadrado entre o ponto corrente e cada uma das K centroides. A menor dessas distâncias define qual centróide será atribuída ao ponto. Após essa verificação, a thread atualiza diretamente o campo cluster_id do ponto correspondente dentro do vetor global de pontos.

Como cada thread trabalha exclusivamente sobre sua própria faixa de pontos e as centroides são somente leitura, não há necessidade de comunicação, mutex ou redução entre threads nessa etapa. Assim, ao final da fase de atribuição, cada ponto já possui seu cluster_id atualizado localmente de forma independente por cada thread.

Fase de Atualização

Na fase de atualização, o algoritmo calcula a posição dos centróides usando as somas e contagens dos pontos atribuídos.

Cada thread possui buffers locais próprios para contagens e somas (local_counts e local_sums). Isso evita condições de corrida, pois cada thread atualiza apenas os seus próprios vetores.

Após cada thread calcular as suas somatórias, todas aguardam em uma barreira. Depois disso, apenas a thread 0 combina os resultados locais, realizando a redução global para formar os novos centróides. Finalmente, todas as threads passam por uma segunda barreira, garantindo que usarão os centróides atualizados na próxima iteração.

OpenMP

Fase de Atribuição

A fase de atribuição ocorre através da função assign_points_to_cluster, que calcula a distância euclidiana entre cada ponto e todos os centróides, para encontrar o mais próximo. Assim, essa operação é independente para cada ponto, e logo é uma fase altamente paralelizável. Além disso, é a parte mais "custosa" do código.

Para a sua paralelização, criamos uma região paralela sobre o laço externo da função, responsável por percorrer o conjunto de pontos. Desse modo, a divisão de trabalho entre as threads ocorre de forma que cada thread fique responsável por um bloco contíguo de iterações de aproximadamente M/T pontos, onde M é o número de pontos e T é o número de threads. Vale ressaltar que o openMP utiliza por padrão a distribuição de carga static, e como não há um considerável desbalanceamento de carga, não foi utilizada nenhuma outra cláusula schedule.

Fase de Atualização

A fase de atualização ocorre através da função update_centroid, que recalcula a posição de cada centróide como a média de todos os pontos atribuídos ao seu cluster. Nossa estratégia foi criar uma região paralela onde cada thread possui seu vetor local de contagens (local_counts) e somas (local_sums).

No primeiro momento, sobre o laço que itera sobre todos os pontos, cada thread atualiza apenas seus próprios vetores locais. No próximo passo, utilizamos a cláusula critical para conferir que apenas uma thread entre no laço por vez, uma vez que esse laço é o responsável por combinar todos os vetores locais das threads nos vetores globais (cluster_counts e cluster_sums). Antes do final da região paralela, liberamos a memória alocada para esses vetores locais. Por fim, a atualização em si dos centróides permanece com a mesma lógica da versão sequencial.

OpenMPI

Fase de Atribuição

Na fase de atribuição, atribuímos clusters aos pontos. Para isso usamos a função "assign_points_to_clusters" que itera por M/[número de processos] (bin_length), sendo M o número de pontos totais. Iteramos pelos pontos dados apenas do ponto "start", que será o início da "chunk" que cada processo ficará encarregado de processar. Durante a iteração calcula-se para cada ponto a distância euclidiana para cada centróide e determina a sua centróide ser a com menor distância euclidiana. Após isso é guardado a melhor centróide de cada ponto em um array dado "best_cluster_list". Quando a iteração acaba é feito um Allreduce para reduzir todas as "best_cluster_list"s em uma só com os dados corretos de cada um dos pontos, assim retornando para cada processo a lista atualizada. Por fim itera-se por todos os pontos atualizando o cluster_id de cada um dos pontos desatualizados.

Fase de Atualização

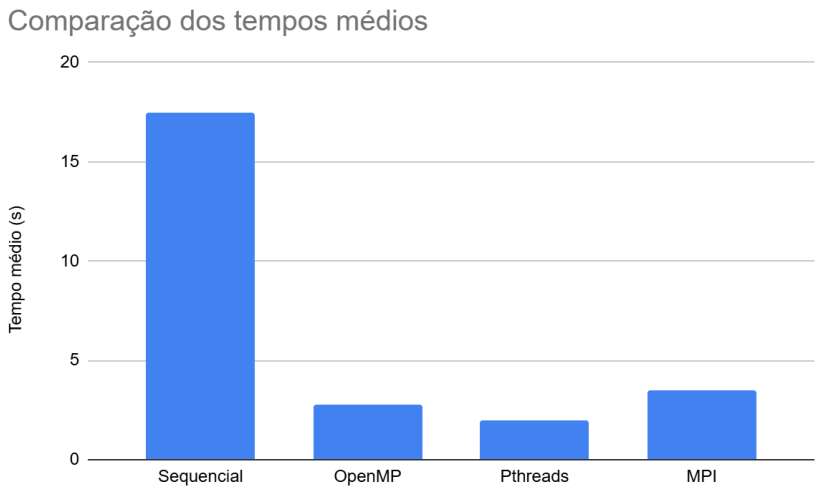
A fase de atualização ocorre com a função "update_centroids", que atualiza as centróides. Esse processo se dá por iterar "bin_length" pontos e, para cada ponto, somar 1 para o contador cluster_counts[num_clusters] do cluster escolhido do ponto. Após atualizarmos o contador somamos as coordenadas dos pontos de cluster_id = n em cluster_sums[n] para todo cluster. Após as iterações faz-se dois Allreduce, um para a soma de todos os cluster_sums e outro para cluster_counts. Por fim itera-se pelos K clusters e D dimensões para atualizar as coordenadas de cada centróide C com o cluster_sums/cluster_counts, fazendo assim com que a coordenada da centróide seja a média das coordenadas dos pontos ligados a tal centróide para todas as centróides.

Análise de Desempenho e Discussão

Discussão dos Resultados

Para analisar o efeito das diversas estratégias de paralelização, todas as versões do algoritmo K-Means foram testadas no mesmo conjunto de dados, o que possibilitou uma comparação direta e consistente. O dataset escolhido foi o mesmo disponibilizado para a avaliação dos códigos, e foram executados com o avaliador também disponibilizado. A versão sequencial foi utilizada como baseline para o cálculo do speedup, sendo executada trinta vezes para garantir uma média consistente. Para as implementações em OpenMP, Pthreads e MPI, o mesmo procedimento foi seguido para assegurar a consistência metodológica. Com base nesses resultados, constatou-se que a execução sequencial teve o maior tempo médio, atingindo 17,5005 segundos. Esse valor reflete o desempenho padrão do algoritmo, sem nenhuma forma de paralelização, tratando cada etapa de maneira linear. Dessa forma, ele define o parâmetro de referência necessário para avaliar o aumento de desempenho oferecido pelas abordagens paralelas.

Versão	Tempo Médio (s)	Speedup
Sequencial	17,5005	1,00x
OpenMP	2,7667	6,33x
Pthreads	1,9887	8,80x
MPI	3,4893	5,02x



A implementação das técnicas de paralelização resultou em uma diminuição considerável no tempo de execução. A versão implementada com OpenMP mostrou-se capaz de utilizar eficientemente vários núcleos, obtendo um tempo médio de 2,7667 segundos. Essa redução significativa demonstra que o modelo baseado em diretivas é capaz de alocar as iterações do algoritmo entre as threads a um custo relativamente baixo. Por outro lado, a implementação com Pthreads alcançou o melhor desempenho de todas, com o menor tempo médio registrado de 1,9887 segundos. Isso sugere que o controle mais direto sobre a criação e sincronização das threads, típico das Pthreads, possibilitou a exploração máxima do paralelismo presente no ambiente de execução. Em contraste, a versão em MPI, embora ainda muito mais rápida que a sequencial, apresentou um tempo maior comparado às outras versões paralelas, com média de 3,4893 segundos. Esse comportamento pode ser explicado pela sobrecarga inerente à comunicação entre processos independentes, o que tende a impactar sistemas de memória compartilhada.

Essas diferenças no tempo de execução têm um impacto direto nos valores de speedup alcançados. Por definição, a versão sequencial mantém um speedup de 1,00x. Por outro lado, o OpenMP obteve um speedup de 6,33x, evidenciando que a utilização de várias threads em um modelo estruturado gerou ganhos significativos. A implementação com Pthreads se sobressaiu ainda mais, obtendo o maior speedup entre todas as abordagens, com 8,80x. Isso demonstra que a menor sobrecarga e o controle mais detalhado sobre as threads levaram a um desempenho geral superior. Por fim, o MPI alcançou um speedup de 5,02x, um resultado positivo, embora inferior aos outros métodos de paralelização empregados neste ambiente específico. Essa diferença reforça que o MPI apresenta desempenho superior principalmente em cenários distribuídos, enquanto em máquinas com memória compartilhada seu overhead de comunicação pode limitar a escalabilidade.

Embora haja diferenças entre os métodos, todas as versões paralelas preservaram a correção total dos resultados, com 30 acertos em 30 execuções. Isso demonstra que os ganhos de desempenho não afetaram a precisão do algoritmo. Em resumo, a análise mostra que a paralelização resultou em melhorias significativas no tempo de execução do K-Means, com a abordagem baseada em Pthreads sendo a mais eficiente no contexto analisado. Os dados mostram que a escolha da estratégia de paralelização afeta diretamente o desempenho final e que, quando usados corretamente, os métodos de baixo nível podem proporcionar vantagens consideráveis.

Overhead e Escalabilidade

A avaliação do overhead e da escalabilidade das diversas estratégias de paralelização possibilita uma compreensão mais acurada do desempenho das implementações quando expostas a variados níveis de carga e paralelismo. No K-Means, cada técnica impõe um custo adicional diferente, que afeta diretamente a eficiência alcançada. O OpenMP exibiu um overhead moderado, uma vez que o ambiente gerencia a criação e sincronização das threads, facilitando o desenvolvimento, embora mantenha algum custo em partes que demandam coordenação entre as threads. Ainda assim, esse overhead não afeta os ganhos alcançados, sugerindo que o OpenMP se ajusta bem ao padrão iterativo e repetitivo do algoritmo.

Entre as técnicas analisadas, a abordagem com Pthreads apresentou o menor overhead. A redução de operações intermediárias e ajustes automáticos é consequência do controle direto sobre a criação e gerenciamento das threads, o que leva a uma diminuição nos custos operacionais. Essa particularidade evidencia a superioridade das Pthreads em termos de tempo de execução e speedup, ressaltando sua eficácia em algoritmos que podem ser paralelizados, como o K-Means. Por outro lado, a implementação com MPI exibiu o maior overhead, devido à exigência de comunicação explícita entre os processos. Nesse modelo de troca de mensagens, em um ambiente de memória compartilhada, surgem custos extras que não estão presentes nas implementações baseadas em threads, o que explica seu desempenho inferior em comparação com outras abordagens paralelas.

Em relação à escalabilidade, Pthreads e OpenMP demonstraram uma maior capacidade de crescimento com o aumento do número de threads, contanto que o hardware forneça núcleos suficientes. O Pthreads geralmente apresenta uma escalabilidade mais linear devido ao seu controle detalhado, ao passo que o OpenMP mantém um bom desempenho, embora possa enfrentar problemas de desempenho se houver muitas sincronizações. Por outro lado, o MPI demonstra sua escalabilidade de forma mais clara em ambientes distribuídos. Quando executado em um único nó, o aumento no número de processos intensifica a troca de mensagens, o que diminui a eficiência marginal. Dessa forma, nota-se que as abordagens baseadas em threads são mais eficientes em sistemas multicore, ao passo que o MPI depende bastante da arquitetura para obter seu melhor desempenho.

Impacto da comunicação e sincronização

A comunicação e a sincronização têm um impacto significativo no desempenho das várias estratégias de paralelização implementadas no algoritmo K-Means. Como é um método iterativo, em que cada etapa depende dos resultados da anterior, a coordenação entre as unidades de processamento se torna imprescindível. Essa dependência impacta diretamente o tempo total de execução, pois operações de comunicação ou sincronização podem causar atrasos, especialmente se forem realizadas com frequência ou se envolverem múltiplos threads ou processos.

Na implementação com OpenMP, o efeito da sincronização é notável, porém moderado. O uso de diretivas simplifica a distribuição das tarefas, mas partes como o cálculo das médias das novas coordenadas dos clusters demandam momentos de junção das threads, o que gera pontos de barreira. Apesar de essas operações terem custo, o OpenMP consegue compensá-lo de forma eficaz graças ao suporte nativo do compilador e ao modelo de memória compartilhada, de modo que a sincronização não afeta significativamente o desempenho geral.

Com Pthreads, a sincronização tem um impacto ainda menor, especialmente porque o programador tem controle direto sobre o momento e a forma como as threads interagem. Isso possibilita a diminuição de chamadas de sincronização desnecessárias e a organização do acesso aos dados de forma mais eficaz. A falta de estruturas intermediárias, típicas de modelos mais abstratos, reduz ainda mais o custo de coordenação. Como resultado, a sincronização impacta o desempenho, porém de maneira mais controlada e menos custosa, o que contribui para o excelente speedup alcançado nessa estratégia.

Por outro lado, entre as técnicas analisadas, o MPI exibe o maior impacto de comunicação. Como cada processo tem seu próprio espaço de memória, o compartilhamento de informações requer a troca explícita de mensagens. No âmbito do K-Means, isso implica que, ao término de cada iteração, os processos devem transmitir seus resultados intermediários para assegurar a consistência global, o que pode resultar em um grande volume de mensagens. Esse custo se torna ainda mais claro em ambientes de memória compartilhada, nos quais o MPI não utiliza o hardware disponível de forma eficiente, gerando mais overhead e desempenho inferior em comparação com as implementações baseadas em threads.

Dessa forma, a análise mostra que o efeito da comunicação e da sincronização difere consideravelmente entre as abordagens. Pthreads e OpenMP conseguem minimizar esse custo por meio do uso compartilhado da memória e da flexibilidade no controle de sincronização, enquanto o MPI enfrenta mais desafios devido à sua necessidade de comunicação explícita e constante. Como consequência, a eficácia das técnicas está diretamente relacionada à maneira como elas tratam esses fatores, o que afeta o desempenho final do algoritmo em cada modelo de paralelização.