Comparação de Execução Sequencial e Paralela Para o Algoritmo de Clusterização K-Means

Michel Bolzon Souza Dos Reis, Luiz Flávio Pereira

[1] Departamento de Informática – Universidade Estadual de Maringá (UEM)
Av. Colombo, 5790 - Zona 7, Maringá - PR - Brazil, 87020-900

ra83558@uem.br, ra91706@uem.br

Resumo. *Neste artigo estaremos fazendo uma comparação de execução do algoritmo K-Means em duas versões, a versão sequêncial e a versão paralela, desta vez utilizando para paralelizar a execução processos ao invés de threads. Foi desenvolvido essas versões do algorimo em Java, fazendo a leitura de arquivos pré-definidos que devem ser selecionados durante a execução do programa.*

1. Introdução

O K-Means é um algoritmo de classificação não supervisionada, este algoritmo tem como idéia principal, o processamento de uma amostra de dados e agrupar essa amostra em N grupos. O algoritmo do K-Means realiza a clusterização dos dados fornecidos, criando assim particionamentos de dados com características semelhantes de uma amostra .

2. Referêncial Teórico

2.1. Threads:

Processo é mais do que um programa, é uma instância de um programa em execução, conceitualmente, cada processo possui sua CPU, mas na prática cada processo ocupa a CPU por um pequeno instante de tempo, sendo assim feita a troca de contexto. Processos são representados no SO(Sistema Operacional) por um bloco de controle de processos(Tabela de Processos), que guarda informações como:

- Id do Processo
- Contador de Programa
- Pilha do Processo, Registradores
- Estado do Processo
- Informações de gerenciamento de memória
- Informações de status de I/O
- Entre outras.

Cada processo possui seu contador de programa, e o caminho dentro do programa que o contador faz é chamado de thread. Muitas threads podem estar presentes no mesmo processo. Devido a essa característica, precisamos criar alguns campos para as threads e uma tabela por thread. Associado a uma thread existe alguns atributos:

- Contador de Programa
- Registradores
- Pilha
- Estado

As threads de um mesmo processo compartilham recursos do processo que às pertence, além de compartilhar o mesmo recursos elas também compartilham o mesmo espaço de endereçamento do processo e váriaveis globais.

2.2. Barreiras:

Em alguns algoritmos iterativos a solução de uma iteração depende do resultado da anterior, para evitar a criação de muitas threads, uma alternativa é criar barreiras e fazer com que todas as threads esperem em um determinado trecho de código, criando assim, uma barreira de sincronização. Existem diversos tipos de barreiras de sincronização, e cada uma delas tem seus casos onde são melhores utilizadas. Alguns tipos de barreiras são:

- Barreira de contador compartilhado
- Barreira de flags e coordenadores
- Barreiras simétricas

A maneira mais simples de especificar os requerimentos de uma barreira é usando um contador compartilhado, usa-se um contador compatilhado entre todas as threads, o contador inicia em 0 e cada thread que chega no contador o incrementa, quando o contador alcança o valor igual ao número de threads, todas podem prosseguir e o contador é resetado.

2.3. Memória Compartilhada:

Processos são implementados de maneira independente, um processo não acessa a memória de outro, porém, algumas vezes é necessária a interação entre processos para realizar tarefas, atender requisições simultâneas, dividir tarefas para aumentar capacidade de computação. Para facilitar a comunicação, o S.O. fornece meios de troca de informação entre processos, Inter-Process Communication (IPC).

Em geral existem duas formas de IPC:

- Suporte a uma forma de espaço de endereçamento compartilhado: **Memória Compartilhada**
- Comunicação via núcleo do SO: **Troca de Mensagens**

A memória compartilhada é um mecanismo onde uma região de memória pode ser acessada por dois ou mais processos, a região deve ser criada e ligada ao processo, essa região se torna parte do endereçamento do processo, as regiões de comunicação e os processos são gerenciados pelo SO, porém, os processos são responsáveis pelo conteúdo da região. Assim como a memória compartilhada entre threads, essa memória tem acesso direto, e também não possui mecanismos automáticos de sincronização:

- Semáforos
- Barreiras
- Mutex

2.4. Memória Distribuida:

Programas para arquiteturas de memória distribuída precisam de interfaces de comunicação com a rede, operações de leitura e escrita análogas às feitas em variáveis compartilhadas, porém com primitivas tão simples algumas vezes seria necessário o uso de busy waiting, operações especiais de rede que incluem sincronização, da mesma forma que semáforos para variáveis compartilhadas. Essas operações especiais podem ser chamadas de passagem de mensagem, processos compartilham canais, cada canal provê um caminho de comunicação entre os processos, programas que usam o passagem de mensagens são chamados de programas distribuídos, os processos podem ser distribuídos entre

processadores de uma arquitetura de memória distribuída, também é possível que um programa possa ser executado como vários processos em um mesmo processador, os canais são implementados como memória compartilhada e não como uma rede de comunicação.

A passagem de mensagem é sempre fluxo único de informação, porém ela pode ser:

- Síncrona
- Assíncrona

A forma mais comum de passagem de mensagem é a passagem de mensagem assíncrona, na passagem de mensagem assíncrona os canais podem ser vistos como semáforos que transportam dados, as primitivas send e receive podem ser vistas como operações V e P, um canal é uma fila de mensagens que foram enviadas e ainda não foram recebidas.

2.5. Envio de Mensagem:

A primitiva send é assíncrona, envia a mensagem e não faz com que o processo que a enviou espere que a mensagem seja recebida, a execução pode continuar. A primitiva sync_send é síncrona, envia a mensagem e faz com que o processo que enviou a mensagem espere até que alguém a receba, a execução fica aguardando a mensagem ser recebida, a passagem de mensagem síncrona tem como principal vantagem a limitação de mensagem no canal de comunicação, reduz o espaço necessário para buffer, no máximo uma mensagem na fila, uma mensagem síncrona pode até manter a mensagem no espaço de memória do processo que envia até que o receptor esteja pronto para recebê-la, Fila constituída apenas pelos endereços dos dados a serem enviados, a passagem de mensagem síncrona tem duas principais desvantagens, redução na concorrência em casos de comunicação, na comunicação, pelo menos um processo irá ser bloqueado

2.6. Recepção de mensagens:

Para receber uma mensagem em um canal um processo usa a primitiva receive, Ao executar a primitiva receive um processo aguarda até que exista pelo menos uma mensagem na fila do canal, Assim que existir uma mensagem na fila do canal essa mensagem é retirada, não é necessário usar busy-waiting para monitorar o canal com a primitiva receive

2.7. Padrão de comunicação Centralizado:

Para o desenvolvimento da aplicação, adotou-se a ideia de possuir um processo mestre e processos cervos. Assim os cervos fariam todo o processamento, as informações atualizadas são enviadas para o processo mestre. Em alguns pontos da execução, teremos o processo mestre totalmente atualizado e os cervos parcialmente atualizados.

Com isso, fez-se necessário a utilização do conceito de Broadcasts, onde o processo mestre envia no canal de comunicação todos os dados atualizados e os demais processos recebem essas informações e se mantém atualizados para a próxima iteração.

Este modelo de trabalho com processamento paralelo em MPI, é conhecido como estrutura de comunicação ”estrela”. Onde as informações são centralizadas em um único processo.

3. Desenvolvimento

A aplicação desenvolvida em Java inicializa realizando a leitura de dois arquivos(base e centróides), o vetor args[] é alimentado na execução e é utilizado para identificar quais arquivos devem ser lidos. Utilizou-se a biblioteca MPJ Express v0.44 para simular os processamentos em MPI.

A coleta dos tempos respeitam a premissa de que é apenas do momento em que os dados estão, realmente, sendo processados pelo MPI. Após coleta dos tempos, a informação é apresentada em tela ao usuário da aplicação e em seguida o sistema fornece um caixa de diálogo onde pode-se selecionar o path onde o arquivo contendo as novas posições do centróides serão armazenadas.

3.1. Arquitetura:

3.1.1. Model:

O sistema foi desenvolvido usando o conceiro de OO através da modelagem de objetos, os seus models são: PontosModel e CoodenadasModel. Essas duas classes representam os objetos da aplicação.

O objeto CoordenadasModel possui uma lista de inteiros responsável por armazenar cada coordenada dos pontos, além disso a classe possui o atributo idCentroide, responsável por vincular os pontos da base com os pontos dos centróides. Já o PontosModel, é uma lista de CoordenasModel simulando assim cada ponto do arquivo lido na inicialização.

3.1.2. Controllers:

O programa possui dois controllers, cada um com sua responsabilidade bem definida. O ArquivoController é responsável por realizar a leitura dos arquivos durante a inicialização dos objetos, impressão dos pontos lidos e também ao finalizar todo o processamento é possível optar escrever um arquivo .data contendo as novas coordenadas do centróide.

A classe PontosControllerMPI contém a lógica responsável por realizar todo o processamento dos arquivos, comunicação em estrela, trocas de mensagens através de sends, receivs e broadcasts. Na aplicação sequencial, a classe recebe o nome de PontosController apenas, ela possui o mesmo papel de processamento do K-Means, contudo, de modo sequencial e sem trocas de mensagens.

3.1.3. Views:

A aplicação nas faz uso de telas elaboras ou interfaces para facilitar o uso ao usuário, o sistema foi desenvolvido com intuíto de obter o maior desempenho possível no processamento do K-Means. Portanto, essa decisão de projeto implica em que o programa seja executado via terminal por linhas de comandos.

3.1.4. Processos

Para balancear as cargas do sistema e dividir o processamento, tornando o algoritmo mais eficiente, foi realizado o uso do MPI para a divisão da carga de trabalho, De acordo com o parâmetro de entrada, é dividido os dados para cada processo, essa divisão ocorre em sua maioria nas estruturas de laços do código, durante a execução dos for é realizado a divisão do processamento, onde a variável que itera no for é incrementada de acordo com a quantidade de processos cervos.

Com este modelo de divisão, obtém-se a certeza de que os processos cervos passarão por todos os pontos da base e dos centróides, sem pular informações.

4. Experimentos

4.1. Hardware

- Modelo Noteboot: Acer Aspire E15 E5-573G-74Q5
- Processador: Intel Core i7-5500U 2.4GHz
- Núcleos 2 e 4 Processadores Lógicos
- Memória RAM: 8GB
- Placa de Vídeo: NVIDIA GeForce 920M with 2GB Dedicated VRAM
- HD: SSD KINGSTONE SA400S37240G 240GB

4.2. Software

- Windows 10 Home Single Language - 64bits - Português

4.3. Experimento 59 Coordenadas

Abaixo segue a execução dos experimentos utilizando um arquivo com os pontos com 59 coordenadas, estamos utilizando 20 centróides. Para cada um dos tempos foram realizados 5 execuções e calculado a média dos tempos.

Execução	Centróides	Base	Tempo de Execução(ms)	Tipo de Execução	Quantidade de Processos	Speedup
1	20	8994	10983	Sequencial	-	-
2	20	8994	8627	Paralelizado	3	1,273
3	20	8994	6953	Paralelizado	4	1,579
4	20	8994	6171	Paralelizado	5	1,779
5	20	8994	7725	Paralelizado	6	1,421

Tabela 1. Speedup dos arquivos de 59 coordenadas.

4.4. Experimento 161 Coordenadas

Abaixo segue a execução dos experimentos utilizando um arquivo com os pontos com 161 coordenadas, estamos utilizando 20 centróides. Para cada um dos tempos foram realizados 5 execuções e calculado a média dos tempos.

Execução	Centróides	Base	Tempo de Execução(ms)	Tipo de Execução	Quantidade de Processos	Speedup
1	20	8991	12806	Sequencial	-	-
2	20	8991	10313	Paralelizado	3	1,241
3	20	8991	7680	Paralelizado	4	1,667
4	20	8991	6218	Paralelizado	5	2,059
5	20	8991	8366	Paralelizado	6	1,530

Tabela 2. Speedup dos arquivos de 161 coordenadas.

4.5. Experimento 256 Coordenadas

Abaixo segue a execução dos experimentos utilizando um arquivo com os pontos com 256 coordenadas, estamos utilizando 20 centróides. Para cada um dos tempos foram realizados 5 execuções e calculado a média dos tempos.

Execução	Centróides	Base	Tempo de Execução(ms)	Tipo de Execução	Quantidade de Processos	Speedup
1	20	8994	25399	Sequencial	-	-
2	20	8994	20786	Paralelizado	3	1,221
3	20	8994	14792	Paralelizado	4	1,717
4	20	8994	11991	Paralelizado	5	2,118
5	20	8994	13409	Paralelizado	6	1,894

Tabela 3. Speedup dos arquivos de 256 coordenadas.

4.6. Experimento 1380 Coordenadas

Abaixo segue a execução dos experimentos utilizando um arquivo com os pontos com 1380 coordenadas, estamos utilizando 20 centróides. Para cada um dos tempos foram realizados 5 execuções e calculado a média dos tempos.

Execução	Centróides	Base	Tempo de Execução(ms)	Tipo de Execução	Quantidade de Processos	Speedup
1	20	8994	17920	Sequencial	-	-
2	20	8994	14616	Paralelizado	3	1,226
3	20	8994	10361	Paralelizado	4	1,729
4	20	8994	8563	Paralelizado	5	2,092
5	20	8994	8695	Paralelizado	6	2,060

Tabela 4. Speedup dos arquivos de 1380 coordenadas.

4.7. Experimento 1601 Coordenadas

Abaixo segue a execução dos experimentos utilizando um arquivo com os pontos com 1601 coordenadas, estamos utilizando 20 centróides. Para cada um dos tempos foram realizados 5 execuções e calculado a média dos tempos.

Execução	Centróides	Base	Tempo de Execução(ms)	Tipo de Execução	Quantidade de Processos	Speedup
1	20	8991	47554	Sequencial	-	-
2	20	8991	36558	Paralelizado	3	1,300
3	20	8991	24815	Paralelizado	4	1,916
4	20	8991	19459	Paralelizado	5	2,443
5	20	8991	21034	Paralelizado	6	2,260

Tabela 5. Speedup dos arquivos de 1601 coordenadas.

4.8. Métricas de Karp-Flatt

Seguem as métricas de Karp-Flatt para auxiliar na conclusão de desempenho da aplicação ao executar com diversos processos.

Qtde Processos	59	161	256	1380	1601
3	0,6783	0,7087	0,7285	0,7234	0,6538
4	0,5110	0,4665	0,4432	0,4378	0,3625
5	0,4526	0,3570	0,3401	0,3475	0,2616
6	0,6444	0,5843	0,4335	0,3825	0,3309

Tabela 6. Métricas de Karp-Flatt para todas as coletas de SpeedUp.

5. Resultados

Podemos identificar que os algoritmos paralelos tiveram uma ligeira melhora em comparação aos algoritmos sequenciais. A partir das métricas de Karp-Flatt podemos concluir que executando com até 5 processos simultâneos, houve sim ganho de performance. Mas ao rodar as informações com 6 processos, podemos identificar que os tempos de execução aumentam, isso é resultado de overheads.

Os ganhos de performance ficaram abaixo do esperado. Ao compararmos o desempenho da aplicação com Threads e com MPI, podemos identificar que o MPI se mostrou mais lento, consumindo mais recursos(RAM) e tempo de execução.

Identificamos também alguns pontos de gargalos e dificuldades ao utilizar a biblioteca MPJ Express, o mesmo se mostrou demasiadamente lento ao realizar as operações de Send, Recv e Bcast. Uma forma de contornar tal situação, foi realizar o mínimo possível de operações MPI, alimentando os dados em vetor e passando-as de uma única vez para seus receptores.

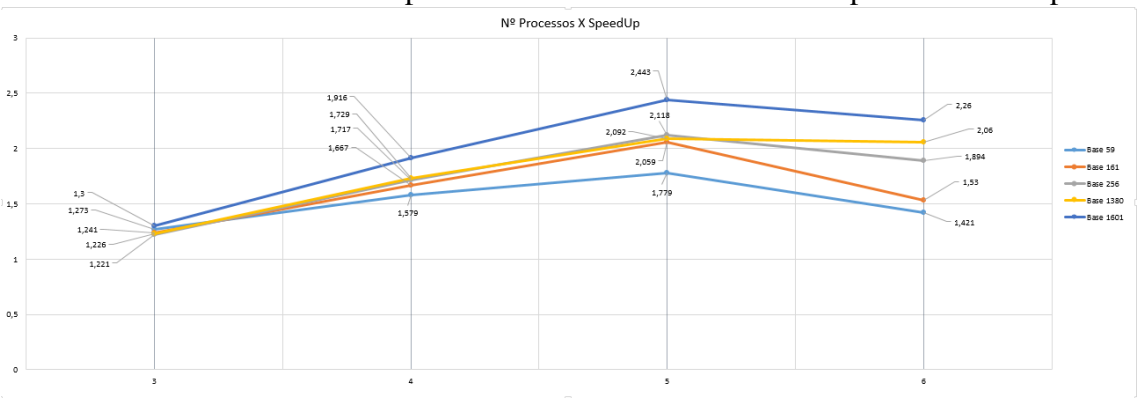

Figura 1. Gráfico representando o Número de processos vs SpeedUp

Através do gráfico acima, identificamos que os arquivos que contém 1380 coordenadas, realizam menos iterações (aproximadamente 21 para cada processo ativo). Já os arquivos com 256 coordenadas há uma quantidade de 160 iterações para cada processo ativo. Com isso, podemos identificar que a variação do Speedup do arquivo maior entre 5 e 6 processos sofre uma menor alteração se comparado ao arquivo de 256 coordenadas, também com essas quantias de processos.

Este fenômeno se dá pelo fato de o arquivo de 1380 coordenadas estar melhor posicionado com relação aos pontos da base fornecidos.