Algoritmo K-Means com Paralelismo

Elio Bolognese Neto[1], **Guilherme Zamberlam Pomini**[1]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)
87.020-900 – Maringá – PR – Brasil

{ra98711, ra99345}@uem.br

Abstract. *This work aims the implementation of Concurrent Programming concepts using a classification algorithm, K-Means. As a way of comparing results and execution time, it was implemented a sequential and a parallel version with variable number of Threads of the algorithm. Through the results obtained it was possible to prove the performance gain of the parallel version, especially the version with four Threads that achieved the best runtime gain.*

Resumo. *Este trabalho visa a implementação de conceitos de Programação Concorrente tendo como base um algoritmo de classificação, o K-Means. Como forma de comparar resultados e tempo de execução, foi implementada uma versão sequencial e uma paralela com número variável de Threads do algoritmo. Através dos resultados obtidos foi possível comprovar o ganho de desempenho da versão paralela, em especial a versão com quatro Threads que conseguiu o maior ganho de tempo de execução.*

1. Introdução

Na computação existem inúmeras formas de encontrar uma solução para um determinado problema. Diferentes formas levam a diferentes usos de recursos, como memória e tempo de execução. Técnicas simples como a de força bruta sacrificam o tempo de execução para gerar uma solução ótima analisando todas as soluções possíveis. A programação dinâmica abre mão de uma parte da memória disponível para salvar soluções anteriores e gerar uma solução ótima com um tempo de execução consideravelmente menor. As abordagens gulosas visam chegar em uma solução através da escolha de caminhos que aparentam ser os que mais vão contribuir para a solução, sacrificando a garantia do resultado ótimo ao utilizar menos memória e com tempo de execução baixo.

Entretanto, por mais rápido que seja o método utilizado, geralmente não utilizam todo o poder do hardware disponível, e com isso problemas com entradas muito grandes podem resultar em tempos de processamento incrivelmente grandes. Os processadores atuais são formados por vários núcleos, que por sua vez possuem suporte para se dividirem em Threads, e normalmente a execução do programa é feita em apenas uma delas.

Assim, neste trabalho será mostrado o quanto de eficiência se ganha ao utilizar várias Threads para executar um programa. Para isso, o problema escolhido foi o K-Means, que será resolvido na linguagem Java com até 10 Threads.

2. Referencial Teórico

Nesta seção será descrito alguns dos conceitos utilizados para a construção do algoritmo analisado neste trabalho.

2.1. K-Means

O algoritmo K-Means baseia-se na minimização de uma medida de custo, isto é, a distância interna entre os padrões de um agrupamento. A minimização do custo garante encontrar um mínimo local da função objetivo, que dependerá do ponto inicial do algoritmo. Esse tipo de algoritmo é chamado de "não-convexo", pois, a cada iteração, diminui o valor da distorção, visto que o resultado final depende do ponto inicial usado pelo algoritmo (DE CASTRO and DO PRADO, 2002).

O problema possui dois conjuntos de dados, os elementos, objetos do universo em questão, e os centroides, responsáveis por agrupar os elementos por características em comum. Para realizar a minimização da distância entre um elemento e um centroide, é necessário alterar a posição desse centroide. Isso é feito verificando qual o melhor centroide para cada elemento. O centroide então é movido para a posição que possui menor distância entre todos os elementos que se agrupam nele. Isto é feito até que nenhum centroide precise ser alterado.

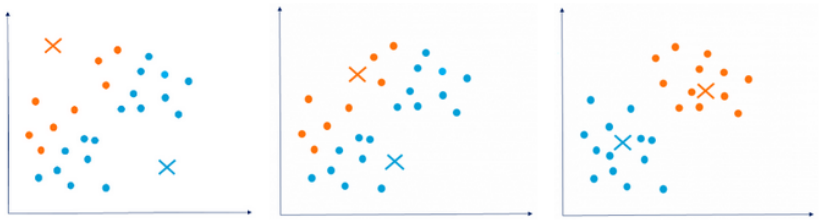

Figura 1. Funcionamento do Algoritmo K-Means

Como pode-se ver na Figura 1, os centroides se movem conforme o grupo deles necessita, encerrando a execução quando não há uma posição melhor existente.

Conseguir agrupar esses dados eficientemente é muito útil para identificar perfis para campanhas de marketing, ou oferecer conteúdos mais relevantes para usuários, ou até mesmo identificar doenças por meio de sintomas semelhantes

2.2. Threads

Os processadores tem como função executar os processos, listas de instruções compostas por linhas de código que informam que passos devem ser executados e em quais momentos devem acontecer. Atualmente o processador é dividido em vários cores físicos, núcleos independentes que funcionam como processadores individuais.

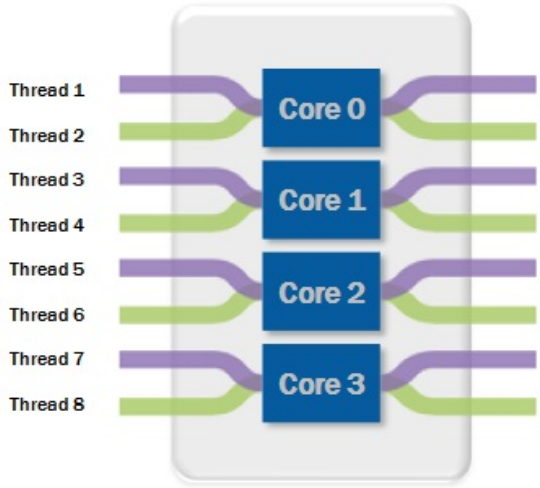

Figura 2. Divisão de Threads em um processador

Esses núcleos, por sua vez, podem ser divididos em uma ou mais Threads lógicas, como mostrado na Figura 2, local onde as linhas de instruções são executadas. Uma Thread é uma linha de código que um processo utiliza para executar um programa. Um processo se divide em várias tarefas e são executadas concorrentemente em Threads diferentes.

O gerenciamento das Threads é feito geralmente pelo próprio sistema operacional (SO), porém é possível que o usuário faça algumas alterações, embora não muito recomendado. Além disso, assim como nem todo processo pode ser dividido em múltiplas Threads, nem todos os processadores são capazes de trabalhar tranquilamente com uma enormidade de Threads.

3. Desenvolvimento

Nesta seção será descrito como que o algoritmo analisado neste trabalho foi implementado.

3.1. Algoritmo K-Means

Para a implementação do algoritmo foram feitas três classes:

1. **Elemento:** armazena uma lista de coordenadas que representam a posição do elemento.
2. **Centroide:** armazena uma lista de coordenadas da posição do centroide e uma lista de elementos que estão no conjunto do centroide.
3. **K-Means:** possui uma lista de elementos e uma lista de centroides, executa o algoritmo em si.

O funcionamento do algoritmo é feito da seguinte forma: para cada elemento é calculado a sua distância para todos os centroides e armazenado o índice do melhor centroide. O cálculo da distância é feito utilizando a distância euclidiana com base nas coordenadas dos centroides e do elemento. Depois da verificação do melhor centroide para cada elemento, os elementos são adicionados nas listas dos seus respectivos centroides.

Com cada centroide tendo a sua nova lista de elementos (que é zerada a cada iteração), é preciso calcular as novas coordenadas de cada centroide. Essa atualização é feita fazendo a média de cada coordenada dos elementos da lista de elementos do centroide. Caso nenhum centroide mude de coordenada, o algoritmo finaliza sua execução pois encontrou um ponto de estabilização, caso contrário o processo é repetido.

3.2. Algoritmo K-Means Paralelo

A versão paralela segue exatamente a mesma estrutura da versão sequencial, com diferença na hora de atribuir os elementos aos centroides. Na versão sequencial, é feito o cálculo de qual é o melhor centroide sequencialmente, seguindo a ordem da lista de elementos. Na versão paralela os elementos da lista foram separados em Threads diferentes, portanto o cálculo é feito de forma paralela para ganho de computação e menor tempo de processamento.

A divisão é feita, por exemplo, no caso de duas Threads, com a primeira metade da lista de elementos na primeira Thread e a segunda metade na segunda Thread. É fácil notar que como o único recurso compartilhado são os centroides, não tendo seus valores alterados, apenas lidos, não existe a necessidade de implementação de estruturas de controle concorrente como Semáforos e Barreiras.

No entanto, somente a criação e execução das Threads por si só não resultaria na paralelização da forma desejada, nada garantiria que as Threads seriam de fato executados antes do próximo passo do algoritmo que é recalcular os centroides. Como forma de garantir uma ”barreira”virtual que garanta a execução das Threads antes do cálculo dos centroides, todas as Threads executam o método Join, forçando a Thread principal do algoritmo esperar a execução de todas as Threads antes de prosseguir o código. Devido ao tratamento com o Join, o algoritmo sequencial e o paralelo geram exatamente os mesmos resultados.

4. Ambiente experimental e Experimentos realizados

- **Software**: O experimento foi realizado no sistema operacional Windows 10 64 bits;
- **Hardware**: Foi utilizado uma máquina física com processador Intel i5-7200U, com 2 núcleos (4 Threads) e 8GB de memória RAM;
- **Algoritmo**: O algoritmo construído recebe como entrada dois arquivos, um contendo os centroides e outro contendo a base de dados, ambos com a mesma quantidade de coordenadas. A versão paralela do algoritmo espera também um valor para a quantidade de Threads, que pode variar entre 2 e 10. É aplicado o processo de K-Means até encontrar uma estabilização nos conjuntos, que se dá quando nenhum centroide tem suas coordenadas alteradas após uma iteração do algoritmo;
- **Experimento**: Os experimentos foram realizados sequencialmente, utilizando a forma sequencial do algoritmo, e paralelamente, utilizando de 2 a 10 Threads. Cada conjunto de bases foi executada 3 vezes para cada configuração (sequencial e variações da quantidade de Threads), como forma de avaliar de forma concisa a execução do algoritmo. Os valores utilizados para as análises foram as médias das 3 execuções.

5. Análise e discussão dos resultados

Com os dados obtidos a partir dos experimentos descritos na seção anterior, foram gerados gráficos para analisar o desempenho do algoritmo.

5.1. Tempo de Execução

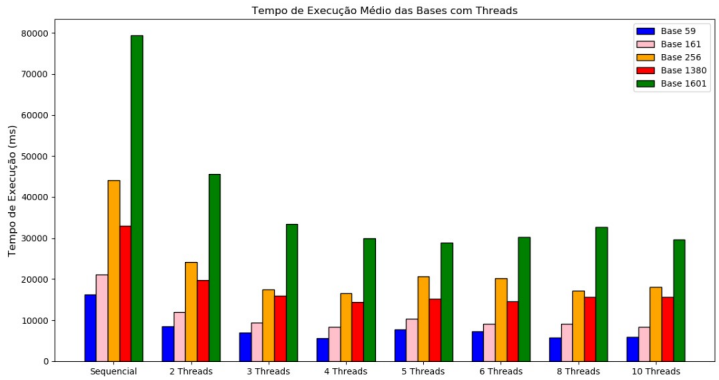

Figura 3. Média de Tempo de execução do algoritmo

Prontamente foi notado na Figura 3 que a base de dados de 256 elementos necessitou de um maior tempo de processamento com relação a base de 1380 elementos. Isso ocorreu pois o estado inicial dos centroides era significativamente pior, acarretando num crescimento não exponencial da razão elementos por tempo de execução. Assim, concluiu-se que essa base é ineficiente para a análise do algoritmo.

Devido ao processador utilizado possuir suporte para apenas 4 Threads, nota-se uma pequena queda no desempenho a partir da quinta Thread, com um nivelamento conforme o aumento de Threads. Essa variação ocorre pois ele não conseguirá executar mais operações paralelamente do que o máximo permitido, porém existem dados em mais locais diferentes, que devem ser retornados para o local de origem antes de eliminar as Threads criadas.

5.2. Speedup

Com as médias dos tempos de execução das Threads foi possível extrair a métrica de Speedup. O cálculo do Speedup mostra qual foi o ganho da versão paralela em comparação com a versão sequencial. O cálculo foi feito seguindo a fórmula:

$$S(t) = \frac{Tempo\ de\ Execucao\ Sequencial}{Tempo\ de\ Execucao\ com\ t\ Threads} \tag{1}$$

Com o Speedup relativo de cada Thread foi possível gerar o gráfico da Figura 4:

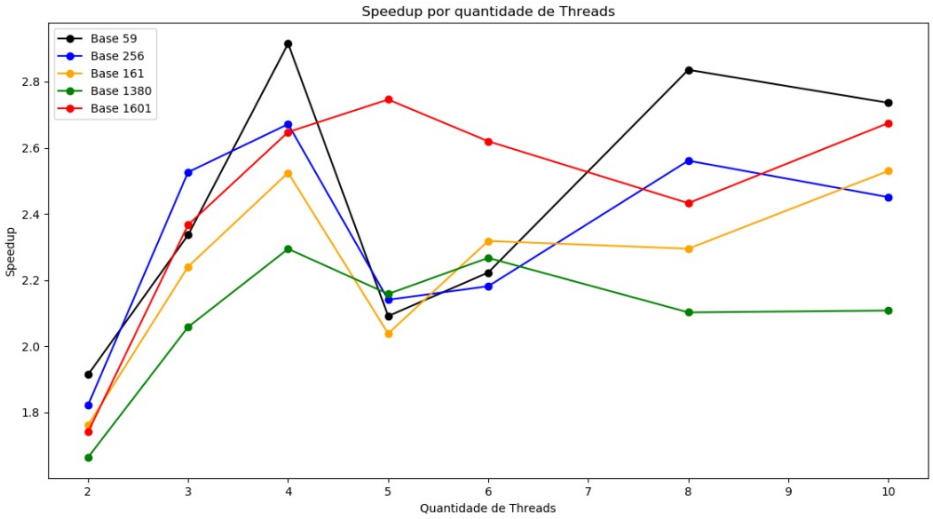

Figura 4. Speedup relativo

É fácil notar que o Speedup foi crescendo conforme a quantidade de Threads foi aumentando, até o pico com 4 Threads que é o limite do processador onde foi realizado os testes. O comportamento após o limite de Threads do computador não é uniforme, com 5 Threads tendo um ganho menor do que com 4 Threads, mas voltando a ter ganho com o aumento das Threads embora nunca chegue ao ganho máximo da execução com 4 Threads.

Com o Speedup é possível notar que a divisão dos elementos nas Threads não é feita de forma exata. Embora a maioria das bases tenha o maior ganho de Speedup com quantidade de Threads múltiplas de 4, a base 1601 tem um ganho maior com múltiplos de 5, sendo a única a ter um desempenho superior com 5 Threads.

5.3. Métrica de Karp-Flatt

A partir do Speedup foi possível aplicar a métrica de Karp-Flatt para obter a fração serial determinada experimentalmente, utilizando a equação

$$e = \frac{1/S(t) - 1/t}{1 - 1/t} \tag{2}$$

onde t é a quantidade de Threads utilizada e $S(t)$ é o Speedup obtido na seção anterior. Foi gerado o gráfico da Figura 5 com os resultados obtidos:

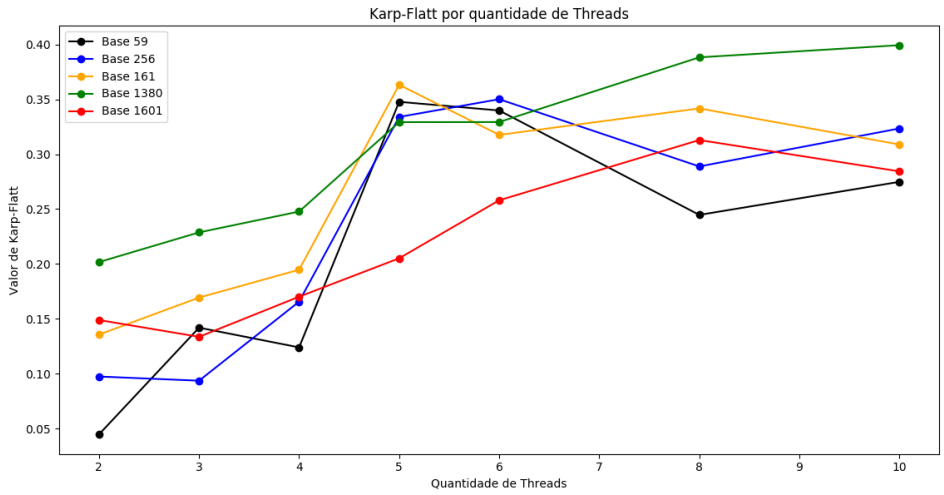

Figura 5. Gráfico da Métrica de Karp-Flatt por Thread

Na maioria das bases, executar o algoritmo com Threads acima do suportado pelo processador causa um aumento de overhead. Além disso, como nem toda divisão dos elementos pela quantidade de Threads será inteira, a Thread que receber os elementos restantes, por menor que seja, resultará também em overhead, como por exemplo na base 59 que trabalha melhor com bases múltiplas de 4.

6. Conclusão

Programação Concorrente é um paradigma de programação que visa melhorar o tempo de execução de algoritmos ao executar tarefas de forma simultânea ao invés de sequencial. A criação de Threads de execução paralela é limitada pela quantidade de núcleos de um computador, formando um gargalo no ganho de tempo de execução.

Os conceitos de Programação Concorrente foram aplicados na implementação do Algoritmo K-Means que trabalha classificando elementos de uma base de dados em conjuntos de acordo com a distância euclidiana de pontos base chamados centroides.

Como forma de experimentar os efeitos da execução paralela do algoritmo K-Means, o mesmo foi executado com uma quantidade diversa de Threads e também sequencialmente para se ter uma base do tempo de execução do algoritmo.

Após os experimentos foi possível observar que o ganho de tempo de execução da versão paralela chega a ser por volta de 50% no melhor cenário com o máximo de Threads suportada pelo processador. Com a análise do Speedup e da Métrica de Karp-Flatt, foi possível verificar o ganho relativo das execuções paralelas, confirmando o aumento do ganho esperado.

Em todos os casos de teste, as versões paralelas se sobressaíram a versão sequencial, melhorando a performance conforme a quantidade de Threads aumentava, até chegar no limite do processador onde o desempenho possui um comportamento igual ou parecido aos seus múltiplos antes do limite.

Referências

ARMANDO ANTONIO MONTEIRO DE CASTRO and PEDRO PAULO LEITE DO PRADO. Algoritmos para reconhecimento de padrões. *Revista Ciências Exatas*, 8 (2002), 2002.