Paralelização do algoritmo K-means com MPI

João Pedro Picolo - GRR20182659

Dezembro de 2020

1 Introdução

De acordo com [1] o algoritmo K-means é um método iterativo e não supervisionado que tem por finalidade classificar dados ao procurar por padrões em um determinado conjunto. Neste trabalho pretende-se paralelizar o algoritmo tradicional utilizando MPI de forma a permitir que o algoritmo possa ser executado em diferente máquinas como forma de paralelismo.

2 Metodologia

Nesta seção pretende-se explicitar qual foi a metodologia utilizada para a elaboração do trabalho, assim como as especificações do computador e das *flags* de compilação utilizadas.

2.1 *Flags* de compilação

De forma a gerar o arquivo executável utilizou-se as *flags* de compilação descritas na Tabela 1.

Flag	Descrição
-Wextra	Gera avisos sobre problemas no código que podem indicar erros, mas não impede a compilação.
-O3	Utiliza nível 3 de otimização feita pelo compilador.
-lm	Liga a biblioteca *math.h* ao programa executável.
-o	Usado para definir o nome do programa executável.

Tabela 1: Descrições das *flags* utilizadas

2.2 Arquitetura

O programa executável foi compilado com o Open MPI 2.1.1 com otimizações em uma máquina com as seguinte configurações: Linux Mint 19 Tara x86_64 Kernel 4.15.0-20-generic com dual core, processador Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz e 4GB de memória DDR3 1600 MHz.

A topologia da máquina utilizada pode ser vista na Figura 1.

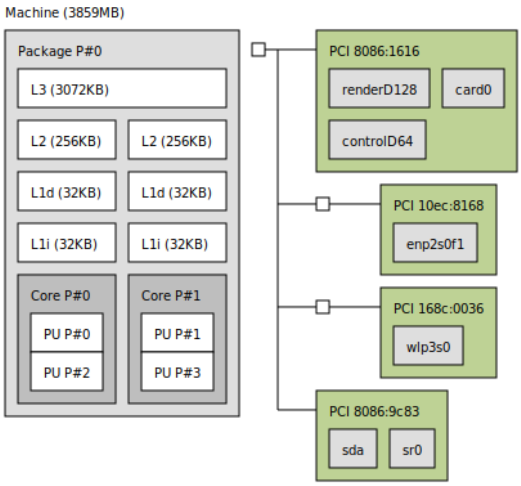

Figura 1: Topologia da máquina utilizada

2.3 Metodologia

Para a medição do tempo total de execução do algoritmo, utilizou-se a diferença entre os valores retornados pela função **MPI_Wtime()** nos trechos de código pertinentes à medição realizada.

Todas as medições de tempo descritas neste trabalho estão em "segundos", seguindo o Sistema Internacional de Unidades, e os valores apresentados são calculados a partir da média aritmética entre 20 iterações do algoritmo analisado.

Para a realização dos testes, como forma de garantir que haja um aumento proporcional do tempo de execução do algoritmo, optou-se por dobrar o número de pontos a serem classificados a cada etapa, mas manteve-se a quantidade de 10 *clusters* utilizados.

3 Algoritmo Sequencial

Neste capítulo pretende-se analisar o comportamento básico do algoritmo sequencial K-means. Algumas variáveis importantes estão descritas na Tabela 2.

Considere o seguinte trecho de código:

```
for (i = 0; i < n; i++) // Inicializa variavel
  cluster[i] = 0;

flips = n; // Incialmente assumimos que havera mudanca
while (flips > 0) { // Repetimos ate que nao hajam mudancas
  flips = 0;

  // Inicializa variaveis com elementos neutros
  for (j = 0; j < k; j++) {
    count[j] = 0;
    for (i = 0; i < DIM; i++)
      sum[j*DIM+i] = 0.0;
  }

  // Para cada numero n, calcula-se a distancia dx ate cada centroide k
  for (i = 0; i < n; i++) {
    dmin = -1; color = cluster[i];

```

```
19      for (c = 0; c < k; c++) {
20        dx = 0.0;
21        for (j = 0; j < DIM; j++)
22          dx += (x[i*DIM+j] - mean[c*DIM+j])*(x[i*DIM+j] - mean[c*DIM+j]);
23
24        // Se for a primeira iteracao ou a distancia de n ate k for menor que
      a distancia de n ate k-1, atualiza variaveis de controle
25        if (dx < dmin || dmin == -1) {
26          color = c;
27          dmin = dx;
28        }
29      }
30
31      // Se houve mudanca de cluster, atualiza variavel de controle para
      outra iteracao e atribui-se em cluster[i] o numero a qual cluster o i-
      esimo n pertence
32      if (cluster[i] != color) {
33        flips++;
34        cluster[i] = color;
35      }
36      }
37
38    // Atualiza count[] com o numero de pontos pertencens ao cluster
39    for (i = 0; i < n; i++) {
40      count[cluster[i]]++;
41      for (j = 0; j < DIM; j++)
42        // Atualiza sum[][] com o somatorio das coordenadas dos pontos
      pertencentes ao cluster
43        sum[cluster[i]*DIM+j] += x[i*DIM+j];
44    }
45
46    // Recalcula a posicao de cada k centroide no espaco
47    for (i = 0; i < k; i++) {
48      for (j = 0; j < DIM; j++) {
49        mean[i*DIM+j] = sum[i*DIM+j]/count[i];
50      }
51    }
52  }
```

3.1 Análise lógica

Considerando N pontos no espaço, o algoritmo funciona do seguinte modo: são selecionados K pontos como os centroides dos *clusters* a serem formados; cada ponto no espaço é atribuído ao *cluster* mais próximo; a média de cada coordenada dos pontos pertencentes à um *cluster* é calculada e estes valores tornam-se as novas coordenadas do centroide correspondente. Este processo se repete até que todos os dados estejam atribuídos ao *cluster* mais próximo, ou seja, até que seja feita uma iteração em que nenhum ponto é reatribuído à um outro *cluster*.

3.2 Análise da execução sequencial

Os tempos médios de execução do algoritmo, seus desvios padrões e a porcentagem de tempo puramente sequencial são descritos na Tabela 3.

Variável	Descrição
k	Número de centroides, indicam a existência de k *clusters*.
n	Número de dados a serem classificados.
DIM	Dimensão do espaço a ser analisado. Valor padrão é 3.
x	Matriz n x DIM, contendo na i-ésima linha as coordenadas (x,y,z) do i-ésimo número, i = [0..n-1].
mean	Matriz k x DIM, contendo na i-ésima linha as coordenadas (x,y,z) dos i-ésimo centroide, i = [0..k-1].
sum	Matriz k x DIM, contendo na i-ésima linha a somatória das coordenadas (x,y,z) de todos os pontos pertencentes ao i-ésimo *cluster*, i = [0..k-1].
cluster	Vetor de tamanho n contendo na i-ésima posição o número do k-ésimo *cluster* ao qual o i-ésimo número pertence, i = [0..n-1] e k = [0..k-1].
count	Vetor de tamanho k contendo na i-ésima posição o número de pontos pertencentes ao i-ésimo *cluster*, i = [0..k-1].
flip	Variável de controle que informa se algum ponto foi reatribuído de *cluster*.

Tabela 2: Descrições das variáveis utilizadas

Entrada	Média	Desvio Padrão	% Puramente Sequencial
1M	5,670	0,038	15,356
2M	9,025	0,016	19,164
4M	25,459	0,506	13,606
8M	71,779	0,143	9,645
16M	101,956	3,079	13,570

Tabela 3: Execução do algoritmo sequencial

Para a obtenção da porcentagem puramente sequencial comparou-se o algoritmo sequencial com o paralelo descrito proposto, os tempo de código dos trechos que puderam ser executados apenas pelo processo *root* no paralelo foram medidos na abordagem sequencial e seus valores foram utilizados para este cálculo junto com o tempo total da execução.

4 Algoritmo Paralelo

Em uma abordagem paralela utilizou-se a biblioteca Open MPI, desta forma é possível criar processos que executem independentemente o trecho de código descrito na seção 3, agora cada processo criado pode se comunicar apenas com um único processo *root*.

Abaixo descreve-se brevemente a abordagem utilizada:

- Inicialmente cada processo recebe seu *rank* identificador e o número de processos em execução, isto é feito através da utilização das funções **MPI_Comm_rank** e **MPI_Comm_size**, respectivamente;
- O processo *root* lê do *stdin* a quantiade de centroides e números, em seguida calcula a quantidade de números que serão analisados por cada processo. Estas informações são repassadas à todos os processos utilizando a função **MPI_Bcast**;

- Cada processo aloca o espaço repassado pelo *root* para cada variável descrita na Tabela 2, adicionalmente o processo *root* aloca mais três variáveis descritas na Tabela 4;
- O processo *root* lê do *stdin* as coordenadas dos centroides e dos pontos a serem classificados, este segundo é armazenado na nova variável *ori_x*;
- Os valores da variável *ori_x* são distribuídos entre todos os processos utilizando a função **MPI_Scatterv** e a matriz *mean* é repassada para os processos utilizando **MPI_Bcast**;
- Cada processo executa o *core* do algoritmo de forma a obter as variáveis *count*, *sum* e *flips* de suas próprias execuções. É feita a soma entre as variáveis de cada processo e esse valor é passado ao processo *root* utilizando a função **MPI_Reduce**. A variável *flips* é somada em todos os processos utilizando **MPI_Allreduce**;
- De posse das informações necessárias o processo *root* recalcula os valores da matriz *mean* e a rapassa novamente para todos os processos utilizando **MPI_Bcast**;
- Os dois passos anterirores são executados até que a soma da variável *flips* seja zero, ou seja, nenhum ponto foi reatribuído em nenhum processo;
- Por fim o processo *root* escreve no *stdout* as novas coordenadas atualizadas dos centroides.

Variável	Descrição
ori_x	Matriz contendo as coordenadas de todos os pontos lidos.
ori_sum	Soma das coordenadas repassadas por cada processo.
ori_count	Soma dos pontos repassados por cada processo.

Tabela 4: Descrições das novas variáveis utilizadas

Note que para a corretude da execução os valores da variável *flips* e *mean* precisam ser atualizados a cada iteração, é menos custoso que apenas um processo realize estes cálculos e repasse à todos os outros.

4.1 Análise da execução paralela

Para o algoritmo paralelo proposto obteve-se os tempos médios de execução descritos na Tabela 5, seus respectivos desvios padrões são descritos na Tabela 6.

É válido notar que em alguns casos o desvio padrão fica muito distante da média. Como todos os testes foram realizados sob as mesmas condições não é possível apontar acertadamente a diferença entre estes valores, entretanto é natural que haja grande interferência por parte de processos que precisam ser realizados pelo Sistema Operacional a acabam por ocupar parte dos recursos disponíveis.

Entrada	1 processo	2 processos	4 processos	8 processos
1M	5,935	3,718	3,807	5,765
2M	9,409	6,082	6,614	10,018
4M	26,280	16,335	16,290	23,276
8M	75,002	44,689	41,871	55,834
16M	105,791	65,109	64,926	92,531

Tabela 5: Execução do algoritmo paralelo

Entrada	1 processo	2 processos	4 processos	8 processos
1M	0,018	0,034	0,013	0,288
2M	0,047	0,053	0,030	0,343
4M	0,267	0,215	0,047	0,278
8M	1,475	0,299	0,225	1,577
16M	1,464	0,383	0,347	0,826

Tabela 6: Desvio padrão do algoritmo paralelo

4.2 *SpeedUp*

Utilizando os dados apresentados na seção 3.2 calculou-se o *SpeedUp* máximo teórico segundo a Lei de Amdahl, conforme descrito na Tabela 7. Na tabela 8 descreve-se o *SpeedUp* máximo teórico segundo a Lei de Gustafson-Barsis, para este cálculo utilizou-se o tempo puramente sequencial (executado apenas pelo processo *root*) durante a execução do algoritmo paralelo proposto.

Entrada	2 processos	4 processos	8 processos	Infinitos processos
1M	1,734	2,738	3,856	6,512
2M	1,678	2,540	3,4167	5,218
4M	1,761	2,841	4,0974	7,350
8M	1,824	3,102	4,776	10,369
16M	1,761	2,843	4,1028	7,369

Tabela 7: *SpeedUp* máximo teórico - Lei de Amdahl

Entrada	2 processos	4 processos	8 processos
1M	1,754	2,663	3,961
2M	1,670	2,463	3,296
4M	1,780	2,751	3,938
8M	1,839	3,031	4,602
16M	1,779	2,749	3,903

Tabela 8: *SpeedUp* máximo teórico - Lei de Gustafson-Barsis

Já com base nos valores coletados e descritos na seção 4.1, obteve-se os *SpeedUps* médios reais, descritos na Tabela 9.

Entrada	1 processo	2 processos	4 processos	8 processos
1M	0,955	1,525	1,489	0,983
2M	0,959	1,484	1,364	0,901
4M	0,969	1,559	1,568	1,094
8M	0,957	1,606	1,714	1,286
16M	0,964	1,566	1,570	1,102

Tabela 9: *SpeedUp* obtido

Nesta etapa nota-se que com a execução de apenas um processo o *SpeedUp* é menor que 1, isto é, o algoritmo ficou mais lento se comparado com o sequencial. Naturalmente este fator é decorrente do *overhead* causado pela paralelização, pois há a necessidade de comunicação inter-processos e computação excessiva.

Também é possível verificar o impacto deste *overhead* durante a comparação do valor máximo real com os teóricos. Além da diferença em relação aos valores previstos pela Lei de Amdahl que desconsidera que o tamanho da entrada aumentaria de acordo com o crescimento de recursos disponíveis, é possível notar também a diferença em relação aos valores gerados pela Lei de Gustafson-Barsis que leva em conta este último fator. Ao compararmos com os valores obtidos em [2] é possível perceber que o *overhead* gerado pela utilização de MPI é maior do que o gerado pela utilização de OpenMP, ao menos em entradas de tamanho pequeno.

4.3 Eficiência e Escalabilidade

A eficiência do algoritmo paralelo proposto é descrita na Tabela 10.

Entrada	1 processo	2 processos	4 processos	8 processos
1M	0,955	0,762	0,372	0,123
2M	0,960	0,742	0,341	0,113
4M	0,969	0,779	0,391	0,137
8M	0,957	0,803	0,429	0,161
16M	0,964	0,783	0,393	0,138

Tabela 10: Eficiência obtida na abordagem paralela

A partir dos dados coletados é possível perceber que o algoritmo não é fortemente escalável visto que sua eficiência cai a medida que se dobra o número de processos e mantêm-se o tamanho da entrada, isto é, os recursos computacionais disponíveis não estão sendo bem aproveitados.

Entretanto nota-se que há a tendência da eficiência aumentar conforme se dobra o tamanho da entrada e se mantém o número de processos, isto indica que o algoritmo proposto é fracamente escalável.

5 Conclusão

O algoritmo apresentado se mostrou uma boa alternativa ao sequencial quando houver a disponibilidade de mais recursos computacionais, isto é, quando não for executado

por apenas um processo pois isso pode deixar a execução mais lenta se comparado com a abordagem sequencial. Utilizando os recursos de apenas uma máquina executando múltiplos processos foi possível observar que o algoritmo apresenta escalabilidade fraca, desta forma conforme aumenta-se o tamanho da entrada o algoritmo possui a capacidade de reduzir significativamente o tempo necessário para a computação.

Referências

[1] S. Na, L. Xumin e G. Yong. "Research on k-means Clustering Algorithm: An Improved k-means Clustering Algorithm". Em: *2010 Third International Symposium on Intelligent Information Technology and Security Informatics*. 2010, pp. 63–67. DOI: 10.1109/IITSI.2010.74.

[2] João Pedro Picolo. "Paralelização do algoritmo K-means com OpenMP". Em: *Trabalho 1 - Programação Paralela* (2020).