Paralelização do algoritmo K-means com OpenMP

João Pedro Picolo - GRR20182659

Novembro 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 de forma a realizar uma computação mais rápida e eficiente.

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.
-fopenmp	Liga a biblioteca do OpenMP 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 GNU C Compiler (gcc) com otimizações em uma máquina com as seguinte configurações: Linux Mint 19 Tara x86_64 Kernel 4.15.0-20-generic com dual core (4 threads) 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.

2.3 Metodologia

Para a medição do tempo total de execução do algoritmo, utilizou-se o comando *time* do *Linux* e para a medição do tempo puramente sequencial do algoritmo utilizou-se a função *omp_get_wtime(void)* da biblioteca *OpenMP*.

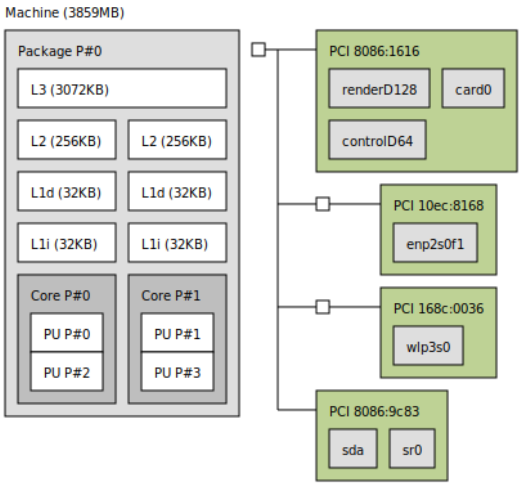

Figura 1: Topologia da máquina utilizada

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

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];

    for (c = 0; c < k; c++) {
      dx = 0.0;
      for (j = 0; j < DIM; j++)
        dx += (x[i*DIM+j] - mean[c*DIM+j])*(x[i*DIM+j] - mean[c*DIM+j]);

      // 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
      if (dx < dmin || dmin == -1) {
        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  }
```

Variável	Descrição
k	Número de centróides, 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 centróide, 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

3.1 Análise lógica

Considerando N pontos no espaço, o algoritmo funciona do seguinte modo: são selecionados K pontos como os centróides 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 centróide 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 metodológica

Os tempos médios de execução do algoritmo acima são descritos na Tabela 3, nela é possível perceber que na iteração de n = 1 milhão o tempo médio gasto em trechos puramente sequencias é de 14,346% enquanto na iteração de n = 5 milhões o tempo médio fica em 3,029%. Esta diferença se da pelo seguinte motivo: conforme a entrada cresce, maior é número de operações e serem executadas no trecho mostrado acima, como este trecho é passível de paralelização o tempo puramente sequencial tende a diminuir.

Entrada	Média	Desv. Padrão	Média Seq.	Desv. Padrão Seq.
n = 1M	6,910	0,090	0,991	0,046
n = 2M	16,037	0,310	1,960	0,550
n = 5M	146,047	0,526	4,424	0,494

Tabela 3: Tempo médio de execução do algoritmo sequencial

4 Algoritmo Paralelo

Em uma abordagem paralela, utilizou-se a biblioteca *OpenMP*. O trecho de código comentado anteriormente, passível de paralelizar, fica da seguinte maneira:

```
#pragma omp parallel for proc_bind(master) num_threads(NUM_THREADS)
    private(i) shared(n)
  for (i = 0; i < n; i++)
    cluster[i] = 0;

flips = n;
while (flips > 0) {
  flips = 0;

   #pragma omp parallel num_threads(NUM_THREADS) private(j,i,dmin,color,c,
   dx) shared(flips,k,n)
   {
       #pragma omp for schedule(guided)
       for (j = 0; j < k; j++) {
           count[j] = 0;
       }

       #pragma omp for schedule(guided)
       for (j = 0; j < k; j++) {
           for (i = 0; i < DIM; i++)
               sum[j*DIM+i] = 0.0;
       }

       #pragma omp for schedule(guided) reduction(+:flips)
```

```
23          for (i = 0; i < n; i++) {
24          dmin = -1; color = cluster[i];
25
26              for (c = 0; c < k; c++) {
27                  dx = 0.0;
28
29                  for (j = 0; j < DIM; j++)
30                      dx += (x[i*DIM+j] - mean[c*DIM+j])*(x[i*DIM+j] - mean[c
    *DIM+j]);
31
32                  if (dx < dmin || dmin == -1) {
33                      color = c;
34                      dmin = dx;
35                  }
36              }
37
38              if (cluster[i] != color) {
39                  flips++;
40                  cluster[i] = color;
41              }
42          }
43      }
44
45    #pragma parallel for proc_bind(spread) reduction(+:sum[:k*DIM])
      num_threads(NUM_THREADS)
46    for (i = 0; i < n; i++) {
47        count[cluster[i]]++;
48        for (j = 0; j < DIM; j++)
49          sum[cluster[i]*DIM+j] += x[i*DIM+j];
50    }
51
52    #pragma parallel for proc_bind(spread) simd num_threads(NUM_THREADS)
53    for (i = 0; i < k; i++)
54      for (j = 0; j < DIM; j++)
55        mean[i*DIM+j] = sum[i*DIM+j]/count[i];
56  }
```

4.1 Análise paralela

A análise lógica e a lista de variáveis seguem o conteúdo descrito na Seção 3, nesta seção será explicitado as escolhas que levaram ao programa paralelo proposto.

Em todas as regiões paralelas foi requisitado ao Sistema Operacional (SO) o mesmo número de *threads* e divide-se os *loops* entre todas as *threads* que foram disponibilizadas pelo SO, note que não necessariamente a quantidade requisitada é disponibilizada.

- 1ª região paralela: utiliza-se da política de afinidade master como forma de manter as operações realizadas sobre o vetor cluster[] o mais próximas possíveis visto que o vetor é de pequeno tamanho e requer bastante acesso à cache, ou seja, é utilizado como forma de evitar o falso compartilhamento.

- 2ª região paralela: a primeira coisa que percebe-se nesta região é a separação entre a inicialização do vetor count[] e da matriz sum[][], esta escolha foi motivada pelo fato de que havendo menos compartilhamento de cache entre estas duas variáveis é possível deixar o código mais rápido ao acessar somente uma variável por vez em uma *thread*. Abaixo descreve-se cada sub-região:

  - 1ª sub-região: neste trecho utilizou-se o *schedule guided* como forma de fazer uma melhor divisão de trabalho entre as *threads* ao permitir que ele seja distribuído dinamicamente e em tamanho variável até que haja um balanceamento na divisão de trabalho.
  - 2ª sub-região: a análise deste trecho segue a da 1ª sub-região.
  - 3ª sub-região: a análise deste trecho segue a da 1ª sub-região com a adição da diretiva *reduction* sobre a variável flip, permitindo que as *threads* criadas escrevam nesta variável com menor quantidade de condições de corrida.

- 3ª região paralela: utiliza-se da política de afinidade esparsa, isto se da pelo fato de que o acesso à matriz x[][] é demorado devido ao seu tamanho então a melhor opção seria manter valores próximos desta matriz na mesma cache. Note que também utilizou-se o *reduction* sobre o vetor sum[] como forma de diminuir as condições de corrida.

- 4ª região paralela: utiliza-se da política de afinidade esparsa. Nesta etapa nota-se o uso da diretiva *simd* como forma de otimizar a solução proposta, visto que permite ao processador realizar mais operações simultâneas.

4.2 Análise metodológica

Para o algoritmo paralelo proposto obteve-se os tempos médios de execução descritos na Tabela 4 e os respectivos desvios padrões são descritos na Tabela 5.

Entrada	1 thread	2 threads	4 threads	8 threads	16 threads
n = 1M	6,850	4,110	3,453	2,995	2,995
n = 2M	15,843	9,266	7,619	6,627	6,592
n = 5M	145,714	79,179	60,057	54,863	54,526

Tabela 4: Tempo médio de execução do algoritmo paralelo

Entrada	1 thread	2 threads	4 threads	8 threads	16 threads
n = 1M	0,027	0,048	0,035	0,006	0,013
n = 2M	0,148	0,062	0,015	0,060	0,039
n = 5M	1,112	0,798	3,560	0,226	0,129

Tabela 5: Desvio padrão sobre o tempo de execução do algoritmo paralelo

Como todos os testes foram realizados sob as mesmas condições, acredita-se que os altos valores de desvio padrão mostrados para a entrada de tamanho 5 milhões tenham sido provocados pela memória cache.

4.3 *SpeedUp*

Com os dados obtidos na Seção 3.2 sabe-se que o tempo passado em trechos passíveis de se paralelizar é 85,654% na menor entrada e 96,97% na maior entrada. Utilizando

Entrada	2 threads	4 threads	8 threads	16 threads	Infinitas threads
n = 1M	1,750	2,799	3,998	5,087	6,993
n = 5M	1,942	3,670	6,612	11,034	33,334

Tabela 6: *SpeedUp* máximo teórico - Lei de Amdahl

estes dados, pode-se calcular aproximadamente o *SpeedUp* máximo teórico segundo a Lei de Amdahl. Estes valores são descritos na Tabela 6.

Já com base nos valores coletados e descritos na Seção 4.2, obteve-se os *SpeedUps* médios reais, descritos na Tabela 7.

Entrada	1 thread	2 threads	4 threads	8 threads	16 threads
n = 1M	1,009	1,681	2,001	2,307	2,307
n = 5M	1,002	1,844	2,432	2,662	2,678

Tabela 7: *SpeedUp* obtido durante os testes

A diferença entre o *SpeedUp* máximo teórico e o real tem motivos. Uma primeira razão é o fato de que a Lei de Amdahl supõe que o tamanho da entrada se mantém conforme aumenta-se os recursos computacionais, mas ao compararmos os resultados pelo tamanho da entrada ainda é possível notarmos a diferença entre a teoria e o mundo real. Isto é causado pelo fato de que a Lei de Amdahl também desconsidera o *overhead* computacional causado pela paralelização do algoritmo.

Desta forma, o *SpeedUp* teórico calculado torna-se pouco realista e deixa de ser um parâmetro a ser alcançável conforme aumenta-se a quantidade de recursos disponíveis.

4.4 Eficiência e Escalabilidade

A eficiência do algoritmo paralelo proposto é descrita na Tabela 8.

Entrada	1 thread	2 threads	4 threads	8 threads	16 threads
n = 1M	1,009	0,840	0,500	0,288	0,144
n = 2M	1,012	0,865	0,526	0,302	0,152
n = 5M	0,989	0,922	0,608	0,333	0,167

Tabela 8: Eficiência obtida durante os testes

A partir dos dados coletados é possível perceber que o algoritmo não é fortemente escalável visto que sua eficiência cai aproximadamente pela metade a medida que se dobra o número de processadores, isto é, os recursos computacionais disponíveis não estão sendo bem aproveitados.

Entretanto, nota-se também que a eficiência não é linear em relação ao número de processadores e o tamanho da entrada, ou seja, quando o número de processadores cresce junto com o tamanho da entrada a eficiência cai indicando que o algoritmo proposto não é fracamente escalável.

5 Conclusão

O algortimo K-means é amplamente utilizado para a classificação de dados em um espaço multidimensional. Sua versão paralelizada, proposta neste trabalho, se mostrou uma boa alternativa no quesito de tempo de execução ao conseguir uma aceleração de até 167,8% ao ser executado com 16 *threads*, entretanto deixou de utilizar aproximadamente 83,3% dos recursos computacionais disponíveis. Vale destacar que o algoritmo também não se demonstrou escalável ao não conseguir utilizar bem os recursos disponíveis.

Desta forma, acredita-se que a melhor opção para a execução do algoritmo proposto seja com a utilização de 4 *threads* pois tendo uma aceleração de até 143,2% consegue-se utilizar mais da metade dos recursos computacionais disponíveis e a eficiência parece aumentar conforme o tamanho da entrada cresce.

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.