K-means - OpenMP

Marcelo dos Santos
marcelouepg@gmail.com

30 de novembro de 2020

1 Introdução

Neste trabalho será feito um estudo da paralelização com openMP do algoritmo de agrupamento *k-means*. Inicialmente serão feitas estimativas da porcentagem do código que pode ser paralelizado e usando a Lei de Amdahl, será estimado o *speedup* máximo para 2, 4, 8 e ∞ processadores. Além disso, com comparações do tempo de execução do algoritmo sequencial e paralelo, serão os medidos *speedups* reais e a eficiência. Finalmente será analisada a escalabilidade forte e fraca do algoritmo.

2 Algoritmo *k-means*

O algoritmo *k-means* é um método de agrupamento de N pontos em um espaço d dimensional em k grupos. A entrada do algoritmo é k, as estimativas iniciais dos centroides dos k agrupamento e um conjunto de pontos X. O funcionamento básico do algoritmo é o seguinte:

- Calcula-se a distância entre cada ponto $\mathbf{x} \in X$ e todos os k centroides. Atribuo ao ponto $\mathbf{x}$ o índice do agrupamento mais próximo.
- Dado que cada $\mathbf{x}$ pertence a um agrupamento, atualizo os centroides dos agrupamentos.
- Repito os 2 passos anteriores até que os índices dos agrupamentos atribuídos a cada $\mathbf{x}$ não mudem mais e o algoritmo convirja.

Antes de mostrar como foi feita a paralelização do algoritmo, vamos rapidamente ver a implementação do algoritmo sequencial.

2.1 Algoritmo sequencial

Inicialmente é feita a inicialização das variáveis *count* e *sum* em zero, pois elas servirão para acumular o número de pontos que pertencem a cada agrupamento e para somar a posição de cada cluster (que será utilizado para calcular a média).

Em seguida temos indicado no código três regiões. A primeira é responsável por calcular a distância entre cada ponto e cada centroide de agrupamento e atualizar a que agrupamento cada ponto pertence. A variável *flip* é responsável por verificar se está ocorrendo mudança nas atribuição de agrupamentos aos pontos $\mathbf{x}$. A segunda região faz a contagem de quantos pontos existem em cada agrupamento e a somatória das posições dos pontos de cada agrupamento. Finalmente na terceira região é calculado o valor da posição média de cada agrupamento utilizando *sum* e *count* calculados na segunda região.

Todo esse trecho de código está dentro de um *while* que executa até que a variável *flip* se mantenha zero. Isso garante que o algoritmo pare após a convergência.

A implementação é mostrada a seguir.

```
flips = 0;
// Inicializao  das variaveis
for (j = 0; j < k; j++) {
   count[j] = 0;
   for (i = 0; i < DIM; i++)
```

```
      sum[j*DIM+i] = 0.0;
}
//Inicio da primeira Regiao
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]);
      if (dx < dmin || dmin == -1) {
         color = c;
         dmin = dx;
      }
   }
   if (cluster[i] != color) {
      flips++;
      cluster[i] = color;
   }
}
//Final da primeira regiao
//Inicio da segunda regio
for (i = 0; i < n; i++) {
   count[cluster[i]]++;
   for (j = 0; j < DIM; j++)
      sum[cluster[i]*DIM+j] += x[i*DIM+j];
}
//Final da segunda regio
//Inicio da terceira regiao
for (i = 0; i < k; i++) {
   for (j = 0; j < DIM; j++) {
      mean[i*DIM+j] = sum[i*DIM+j]/count[i];
   }
}
//Final da terceira regiao
```

2.2 Algoritmo paralelo

As partes mais custosas do algoritmo são a primeira e a segunda região, devido ao laço que varia de 0 a n (N). Assim se por exemplo $n = 10^5$ e o algoritmo executa 10^2 iterações, então cada um dos laços executará 10^7 vezes, sem falar dos laços internos. A primeira tentativa de paralelização (que chamaremos de versão 1) foi na primeira região. A estratégia utilizada foi dividir o laço mais externo (em i) entre cada uma das *threads*. A divisão foi feita por blocos mantendo dados contíguos com a mesma *thread*, ou seja, o primeiro grande bloco fica com a *thread*$= 0$, o segundo com a *thread*$= 1$ e assim por diante. O índice i agora varia entre as variáveis min e max. A maioria das variáveis utilizadas nesta região tornaram-se privadas, pois cada *thread* atualiza seus valores concomitantemente. Esta região foi a mais fácil de ser paralelizada, pois apenas a variável $flip$ teve que passar a ser o vetor $flip\_par[ID]$, onde cada componente é utilizada por uma *thread*. Ao final da região paralela, as componentes deste vetor são acumuladas em $flip$.

O código é mostrado a seguir.

```
//primeira regiao paralela
  #pragma omp parallel num_threads(NUM_THREADS)
  {
     int nthreads=omp_get_num_threads();
     int ID=omp_get_thread_num();
     int tam_p_thread=n/nthreads;
     int min, max,ip,colorp,cp,jp;
     double dmin, dx;
     min=tam_p_thread*ID;
```

```
      if(ID!=nthreads-1){max=tam_p_thread*(ID+1);}
      else{max=n;}
      if(ID==0){num_threads=nthreads;}

      for (ip = min; ip < max; ip++) {
         dmin = -1; colorp = cluster[ip];
         for (cp = 0; cp < k; cp++) {
            dx = 0.0;
            for (jp = 0; jp < DIM; jp++)
               dx += (x[ip*DIM+jp] - mean[cp*DIM+jp])*(x[ip*DIM+jp] - mean[cp*DIM+jp]);
            if (dx < dmin || dmin == -1) {
               colorp = cp;
               dmin = dx;
            }
         }
         if (cluster[ip] != colorp) {
            flip_par[ID]++;
            cluster[ip] = colorp;
         }
      }
   }
   //final da primeira regiao paralela
   for (i=0;i<num_threads;i++){flips=flips+flip_par[i];flip_par[i]=0;}
   //inicio segunda regiao paralelizavel
   for (i = 0; i < n; i++) {
      count[cluster[i]]++;
      for (j = 0; j < DIM; j++){
         sum[cluster[i]*DIM+j] += x[i*DIM+j];
      }
   }
   //final da segunda regiao paralelizavel
   for (i = 0; i < k; i++) {
      for (j = 0; j < DIM; j++) {
         mean[i*DIM+j] = sum[i*DIM+j]/count[i];
      }
   }
```

Além da primeira região, podemos também paralelizar a segunda região, pois também possui um laço em i variando de 0 a n (chamaremos de versão 2 do código paralelizado). Para isso, a variável *count* que tinha tamanho k passou a se chamar *countp* e tem tamanho $k \times NUM\_THREADS$. Analisando como uma matriz, o número de colunas seria k e cada linha seria utilizada por uma *thread*. Outra variável que foi alterada foi *sum* que passou a se chamar *sump* e ter tamanho $DIM \times k \times NUM\_THREADS$. Ao final da região paralela, estas variáveis precisam ser agrupadas nos vetores *count* e *sum*. A paralelização da segunda região é mostrada a seguir.

```
sump = (double *)malloc(sizeof(double)*DIM*k*NUM_THREADS);
countp = (int *)malloc(sizeof(int)*k*NUM_THREADS);
    .
    .
    .
// incio  da segunda regio paralela
#pragma omp parallel num_threads(NUM_THREADS)
{
   int nthreads=omp_get_num_threads();
   int ID=omp_get_thread_num();
   int tam_p_thread=n/nthreads;
   int min, max,ip,jp;
   min=tam_p_thread*ID;
   if(ID!=nthreads-1){max=tam_p_thread*(ID+1);}
   else{max=n;}
```

```
      if(ID==0){num_threads=nthreads;}

      for (ip = 0; ip < n; ip++) {
         countp[k*ID+cluster[ip]]++;
         for (jp = 0; jp < DIM; jp++){
            sump[DIM*k*ID+cluster[ip]*DIM+jp] += x[ip*DIM+jp];
         }
      }
   }
   //final da segunda regiao paralela
   for (i=0;i<num_threads;i++){
      for (j=0;j<k;j++){
         count[j]=count[j]+countp[k*i+j];
         countp[k*i+j]=0;;
         for (l = 0; l < DIM; l++){
            sum[j*DIM+l]=sum[j*DIM+l]+sump[DIM*k*i+j*DIM+l];
            sump[DIM*k*i+j*DIM+l]=0;
         }
      }
   }
```

Foram efetuados testes comparando o ganho no tempo de execução com as duas versões de paralelização citadas acima, mas verificou-se que com a versão 2 (paralelização da primeira e da segunda região) não houve melhora significativa no tempo de execução do algoritmo. Na verdade em alguns casos, o tempo de execução foi até pior que da versão 1. Por esse motivo, optou-se pela versão 1, ou seja, mantendo a paralelização apenas na primeira região.

3 Metodologia

O primeiro passo antes de efetuar os experimentos e obter as medidas de desempenho é medir a porcentagem do tempo de execução que é devido à parte sequencial e a porcentagem que é devido ao pedaço paralelizável. Dada a porcentagem de código sequencial β, e o número de processadores p, a lei de Amdahl diz que o *speedup* S é dado por

$$S = \frac{1}{\beta + (1-\beta)/p}. \tag{1}$$

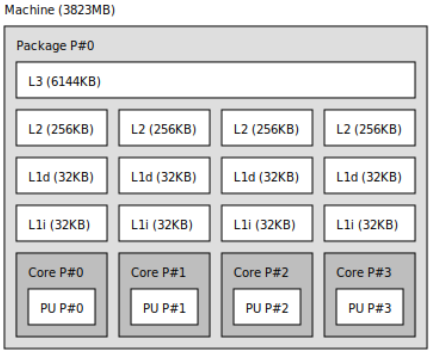
Assim com a estimativa de β podemos estimar S_t (*speedup* teórico) para diversos processadores e comparar com o experimental S_e. Para medir os tempos de execução foi utilizada uma função que fornece o tempo do relógio em pontos específicos do código. Assim foi possível medir o tempo total de execução t_t e o tempo total das regiões sequenciais t_s. Com isso foi estimado β pela razão t_s/t_t.

O *speedup* S_e foi obtido pela razão dos tempos de execução do algoritmo sequencial e paralelo. A eficiência foi obtida dividindo S_e pelo número de processadores p utilizados.

Para realizar os experimento, foram feitos testes variando o número de pontos N e o número de processadores p e medindo o *speedup* e a eficiência. Todos os experimentos foram repetidos 30 vezes e foi calculado um valor médio para se obter estimativas mais fidedignas. Os valores para k foram $k = 10$ e $k = 100$.

A configuração da máquina utilizada é mostrada a seguir.

```
Arquitetura:              x86_64
CPU(s):                   4
Thread(s) per ncleo:      1
Modelo:                   Intel Core i3-8100 CPU 3.60GHz
cache de L1d:             32K
cache de L1i:             32K
cache de L2:              256K
cache de L3:              6144K
Memoria:                  3.7 GiB
Sistema operacional:      Linux Mint 19.1 Cinnamon
gcc version:              7.5.0
```



A compilação foi feita da seguinte maneira

```
gcc -o kmeans_par -fopenmp kmeans_par.c -lm -O3 -Wall
```

4 Resultados

Durante a estimativa de β, verificou-se que os valores tinham uma pequena variação com N. A tabela (1) mostra os valores obtidos juntamente com o desvio padrão.

N	10000	20000	40000	80000	160000
β	0,0281	0,0338	0,0314	0,0179	0,0169
σ_β	0,0018	0,0016	0,0061	0,0028	0,0015

Tabela 1: Valores de β médio e devio padrão σ_β obtidos com o algoritmo sequencial para $k = 100$ e diversos valores de N.

Apesar das variações nos valores de β da tabela (1), os valores de S_t encontrados apresentam pouca variação com N, pois o parâmetro β aparece na Lei de Amdahl no denominador somado com uma constante. A tabela (2) mostra os valores encontrados para S_t e S_e para diversos valores de N e processadores p, considerando $k = 100$.

N	p	2	4	8	∞
10000	S_t	1,9453	3,6890	6,6851	35,5872
	S_e	1,9141	3,4013	2,5602	-
	σ_e	0,0169	0,4078	0,2288	-
20000	S_t	1,9346	3,6317	6,4694	29,5858
	S_e	1,9291	3,5947	2,9262	-
	σ_e	0,0088	0,0545	0,1493	-
40000	S_t	1,9391	3,6556	6,5585	31,8471
	S_e	1,9095	3,6468	3,0665	-
	σ_e	0,0256	0,0192	0,3131	-
80000	S_t	1,9648	3,7961	7,1092	55,8659
	S_e	1,9298	3,7075	3,2384	-
	σ_e	0,0230	0,0760	0,2657	-
160000	S_t	1,9668	3,8070	7,1537	59,1716
	S_e	1,9287	3,7449	3,5288	-
	σ_e	0,0129	0,0258	0,1120	-

Tabela 2: Valores obtidos para os *speedup* teórico e experimental e desvio padrão σ_e dos valores S_e para diversos valores de N, p e $k = 100$.

De forma geral percebe-se que que os valores obtidos para S_e estão próximos dos limitantes teóricos S_t obtidos pela lei de Amdahl, indicando que a estratégia de paralelização adotada foi satisfatória. Outro resultado importante que deve ser mencionado é que como a máquina utilizada possui apenas 4 processadores, quando utilizamos 8 *threads* no algoritmo paralelo, obtemos um S_e bem abaixo de S_t, como era de se esperar. Além disso, quando utilizamos 8 *threads* obtemos um resultado pior que quando utilizamos apenas 4. Isso ocorre porque recursos extras são alocados mas não são utilizados de maneira eficiente devido ao número de processadores limitado em 4.

Podemos também analisar a taxa com que o *speedup* S_e aumenta quando dobramos o numero de processadores. Para $N = 10000$ obtemos $3,4013/1,9141 = 1,78$. Isso indica que estamos abaixo do caso ideal em que dobramos o *speedup* sempre que dobramos o número de processadores. Na verdade a própria lei de Amdahl mostra que o *speedup* aumenta numa taxa menor que 2 quando dobramos o número de processadores. Para ver isso considere o *speedup* S_p para um β e p qualquer:

$$S_p = \frac{1}{\beta + (1-\beta)/p}. \tag{2}$$

Considere agora que dobramos o número de processadores. Neste caso S_{2p} é

$$S_{2p} = \frac{1}{\beta + (1-\beta)/(2p)}. \tag{3}$$

Calculando a razão S_{2p}/S_p encontramos

$$\frac{S_{2p}}{S_p} = 2\left(\frac{p\,\beta + 1 - \beta}{2\,p\,\beta + 1 - \beta}\right). \tag{4}$$

Fazendo $\beta \to 0$, obtemos $S_{2p}/S_p = 2$. Assim o *speedup* só aumenta na mesma taxa que p quando a porcentagem de tempo gasta na parte sequencial for desprezível comparada ao total.

Os resultados obtidos para a eficiência são mostrados na tabela (3). Apesar dos testes terem sido efetuados com apenas 2 e 4 processadores, percebe-se que quando dobramos o número de processadores, mantendo N pequeno e constante, a eficiência tem uma queda considerável, sugerindo que o algoritmo não possui escalabilidade forte. Por outro lado, quando dobramos p e N, percebe-se que a eficiência decai numa taxa mais lenta, como pode ser visto na diagonal em negrito com os valores 1, $0,9645$ e $0,9117$ sugerindo uma escalabilidade fraca. Isso quer dizer que quando o número de processadores for muito grande, mas a entrada também for proporcionalmente grande, será possível manter uma eficiência maior que aquela obtida para N fixo. Ainda na tabela (3) percebe-se que para $p = 8$ a eficiência cai muito e fica abaixo de 50% como já era esperado, pois a máquina possui apenas 4 processadores.

Eficiência				
	Processadores			
N	1	2	4	8
10000	**1**	0,9570	0,8503	0,3200
20000	1	$\mathbf{0,9645}$	0,8986	0,3657
40000	1	0,9547	$\mathbf{0,9117}$	0,3833
80000	1	0,9649	0,9268	0,4048
160000	1	0,9643	0,9362	0,4411

Tabela 3: Eficiência para diversos valores de N, p e $k = 100$.

Uma pergunta que surge neste ponto é se a escalabilidade se mantém para qualquer valor de k. As figuras (1) e (2) mostram a eficiência como função de N para 2 e 4 processadores para $k = 100$ (primeira figura) e $k = 10$ (segunda figura). De forma geral, percebe-se que a eficiência aumenta com N, mas quando aumentamos p obtemos valores menores de eficiência. Mas o ponto importante aqui é que quando k é grande ($k = 100$), a eficiência rapidamente alcança valores satisfatórios (acima de 90% para $N > 4000$), mas quando k é menor ($k = 10$), o aumento da eficiência com N é bem mais lento, pois quando dobramos p e aumentamos N na mesma proporção (ou numa proporção até maior), a eficiência continua muito

baixa (abaixo de 80%). Assim podemos afirmar que para $k = 10$ o algoritmo já não é tão escalável como é para $k = 100$.

A tabela (2) mostra o *speedup* e a eficiência para dados de tamanho da ordem de 10^6, mostrando uma eficiência máxima de $96,81\%$ para $p = 2$ e $84,94\%$ para $p = 4$.

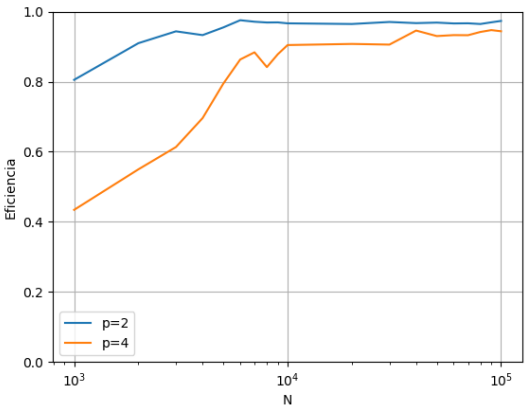


Figura 1: Eficiência como função de N para 2 e 4 processadores e $k = 100$.

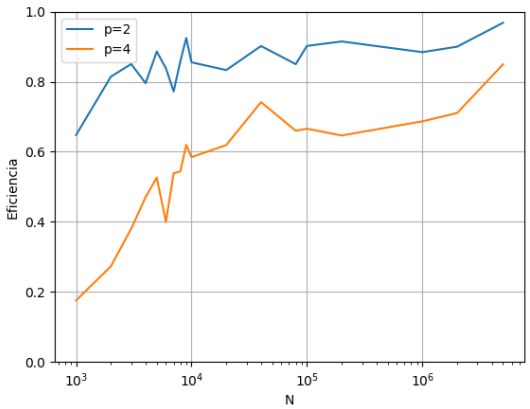


Figura 2: Eficiência como função de N para 2 e 4 processadores e $k = 10$.

N	p	2	4
1.10^6	$S_e(\sigma_e)$	1,7685 (0,0155)	2,7472 (0,0097)
	E	0,8842	0,6868
2.10^6	$S_e(\sigma_e)$	1,7998 (0,0107)	2,8418 (0,0253)
	E	0,8999	0,7104
5.10^6	$S_e(\sigma_e)$	1,9363 (0,0125)	3,3978 (0,0163)
	E	0,9681	0,8494

Tabela 4: Valores obtidos para os *speedup* S_e e eficiência E para alguns valores de N e p e $k = 10$.

5 Conclusão

Neste trabalho foi possível obter os limites teóricos para o *speedup* estabelecidos pela lei de Amdahl. Os valores experimentais também foram medidos e de maneira geral pode-se dizer que chegaram muito próximos dos limites teóricos. Verificou-se também que quanto menor for a parte sequencial do código, maior é o *speedup* alcançável. Por último, os teste realizados indicam que o algoritmo não possui escalabilidade forte, pois a eficiência diminui consideravelmente quando aumentamos o número de processadores de 2 para 4 mantendo N constante. Por outro lado, verificou-se que dependendo do valor de k, podemos ter escalabilidade fraca como visto para $k = 100$. Mas para $k = 10$, quando dobramos o número de processadores e aumentamos N, continuamos com valores muito baixos para a eficiência. Para afirmar com certeza estes resultados, deve-se efetuar mais testes utilizando 8 e 16 processadores e verificando se a eficiência mantém estas taxas de decaimento.