K-means - MPI

Marcelo dos Santos
marcelouepg@gmail.com

16 de dezembro de 2020

1 Introdução

Neste trabalho será feito um estudo da paralelização com MPI 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 medidos os *speedups* reais e a eficiência. Também será analisada a escalabilidade forte e fraca do algoritmo e por último será feita uma breve comparação do *overhead* do OpenMP e MPI.

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;
   }
    //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;
      }
   }
   //Segunda regiao
   for (i = 0; i < n; i++) {
      count[cluster[i]]++;
      for (j = 0; j < DIM; j++)
         sum[cluster[i]*DIM+j] += x[i*DIM+j];
   }
   //Terceira regiao
   for (i = 0; i < k; i++) {
      for (j = 0; j < DIM; j++) {
         mean[i*DIM+j] = sum[i*DIM+j]/count[i];
      }
   }
```

2.2 Algoritmo paralelo

O algoritmo com MPI executa quase que inteiramente em paralelo e isso é feito colocando MPI_init logo no início do código e MPI_finalize logo antes do return 0. Algumas partes devem ser executadas exclusivamente de forma sequencial e isso é feito com um if(my_rank==0){...}, onde my_rank é uma variável utilizada para enumerar e identificar o processo corrente.

O algoritmo funciona da seguinte maneira: o processo 0 lê k e n e grava em $kn[0]$ e $kn[1]$ respectivamente e imediatamente passa essas informações a todos os processos por meio de MPI_Bcast. Estas informações serão utilizadas para saber o tamanho de cada bloco que cada processo vai trabalhar e para fazer as alocações necessárias. Em seguida o processo 0 lê os vetores x e mean. O vetor mean é passado para todos os processos por um MPI_Bcast e o vetor x é dividido em n_procs partes e distribuído para os processos por meio do MPI_Scatter. Nesta etapa cada processo entra no laço e inicia sua parte no trabalho. As variáveis cluster, sum e count são calculadas dentro de cada processo. Após isso é utilizada a função MPI_Allreduce que agrega as variáveis count, sum e flips nas variáveis global_count, global_sum e global_flips respectivamente e atualiza estas variáveis a todos os processos por meio de um *Broadcast* (interno ao MPI_Allreduce). Após isso o laço itera várias vezes e só para quando global_flips é igual a zero.

Observe que as partes mais custosas do algoritmo são a primeira e a segunda região, devido ao laço que varia de 0 a n (chamaremos de N ao longo do texto) e como estas regiões foram paralelizadas com êxito, será visto no resultados uma diminuição no tempo de execução como esperado.

O código paralelizado é mostrado a seguir.

```
int main(int argc, char **argv) {
   int my_rank,n_procs;
   MPI_Init(&argc, &argv);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
int kn[2];
int i,c,k,j, n,*cluster,*count,*global_count,flips,global_flips,color;
double *x, *x_i,*mean,*sum,*global_sum,dx,dmin;
int ret;
if(my_rank==0){
   ret=scanf("%d", &kn[0]);
   ret=scanf("%d", &kn[1]);
}
MPI_Bcast(kn, 2, MPI_INT, 0, MPI_COMM_WORLD);
k=kn[0];n=kn[1];
int tam_p_proc=n/n_procs+1;
x = (double *)malloc(sizeof(double)*DIM*tam_p_proc*n_procs);

int min, max;
min=tam_p_proc*my_rank;
if(my_rank!=n_procs-1){max=tam_p_proc*(my_rank+1);}else{max=n;}

x_i = (double *)malloc(sizeof(double)*DIM*(tam_p_proc));
mean = (double *)malloc(sizeof(double)*DIM*k);
sum= (double *)malloc(sizeof(double)*DIM*k);
global_sum= (double *)malloc(sizeof(double)*DIM*k);
cluster = (int *)malloc(sizeof(int)*n);
count = (int *)malloc(sizeof(int)*k);
global_count = (int *)malloc(sizeof(int)*k);

if(my_rank==0){
   for (i = 0; i<k; i++)
      ret=scanf("%lf %lf %lf", mean+i*DIM, mean+i*DIM+1, mean+i*DIM+2);
   for (i = 0; i<n; i++)
      ret=scanf("%lf %lf %lf", x+i*DIM, x+i*DIM+1, x+i*DIM+2);
}

MPI_Bcast(mean, DIM*k, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(x, DIM*tam_p_proc, MPI_DOUBLE, x_i,DIM*tam_p_proc,MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (i = 0; i<n; i++){cluster[i] = 0;}
free(x);
global_flips=n;
while (global_flips>0) {
   flips = 0;
   global_flips=0;
   for (j = 0; j < k; j++) {
      count[j] = 0;
      for (i = 0; i < DIM; i++){
         sum[j*DIM+i] = 0.0;
         global_sum[j*DIM+i] = 0.0;
      }
   }
   for (i = min; i < max; i++) {
      dmin = -1; color = cluster[i];
      for (c = 0; c < k; c++) {
         dx = 0.0;
         for (j = 0; j < DIM; j++)
            dx += (x_i[(i-min)*DIM+j] - mean[c*DIM+j])*(x_i[(i-min)*DIM+j] - mean[c*DIM+j]);
         if (dx < dmin || dmin == -1) {
            color = c;
            dmin = dx;
         }
      }
      if (cluster[i] != color) {
```

```
            flips++;
            cluster[i] = color;
         }
      }
      for (i = min; i < max; i++) {
         count[cluster[i]]++;
         for (j = 0; j < DIM; j++){
            sum[cluster[i]*DIM+j] += x_i[(i-min)*DIM+j];
         }
      }
      MPI_Allreduce(count, global_count,k,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
      MPI_Allreduce(sum, global_sum,k*DIM,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
      MPI_Allreduce(&flips, &global_flips,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
      for (i = 0; i < k; i++) {
         for (j = 0; j < DIM; j++) {
            mean[i*DIM+j] = global_sum[i*DIM+j]/global_count[i];
         }
      }
   }
   if(my_rank==0){
      for (i = 0; i < k; i++) {
         for (j = 0; j < DIM; j++)
            printf("%5.2f ", mean[i*DIM+j]);
         printf("\n");
      }
   }
   MPI_Finalize();
   return(0);
}
```

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

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

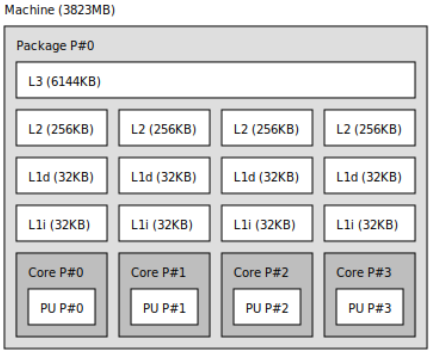
Assim com a estimativa de β podemos estimar S_t (*speedup* teórico) para diversos processadores e comparar com o experimental S_e. Para medir os tempos de execução foi utilizada uma função que fornece o tempo do relógio em pontos específicos do código. Assim foi possível medir o tempo total de execução 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 testados para k foram $k = 10$ e $k = 100$. Também será feita uma breve comparação do *overhead* do OpenMP e MPI para diversos valores de N com $p = 4$ e $k = 10$.

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 e execução foram feitas da seguinte maneira

```
mpicc -o kmeans_MPI kmeans_MPI.c -lm -Wall -O3
mpirun -np 4 ./kmeans_MPI < dados.txt
```

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	0,0272	0,0321	0,0269	0,0143	0,0122
σ_β	0,0011	0,0027	0,0053	0,0021	0,0019

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 na Tabela 1, os valores do *speedup* teórico 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 o *speedup* teórico S_t e experimental S_e para diversos valores de N e processadores p, considerando $k = 100$. De forma geral percebe-se 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 processos no algoritmo paralelo, obtemos um S_e bem abaixo de S_t, como era de se esperar. Além disso, quando utilizamos 8 processos 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 $3,0694/1,7070 = 1,79$. Isso indica que estamos abaixo do caso ideal em que dobramos o *speedup* sempre que dobramos o número de processadores.

A lei de Amdahl prevê que para $\beta > 0$, o *speedup* se aproxima assintoticamente de $1/\beta$ quando $p \to \infty$. Neste caso, a eficiência que é dada por S/p tende a zero quando p é muito grande. Assim é de se esperar que para $\beta > 0$ o algoritmo não possua escalabilidade forte.

N	p	2	4	8	∞
10000	S_t	1,9470	3,6982	6,7204	36,7647
	S_e	1,7070	3,0694	2,1224	-
	σ_e	0,1570	0,3068	0,2899	-
20000	S_t	1,9378	3,6486	6,5322	31,1526
	S_e	1,9263	3,4947	2,3763	-
	σ_e	0,0330	0,1719	0,2590	-
40000	S_t	1,9476	3,7013	6,7323	37,1747
	S_e	1,9410	3,6971	2,8814	-
	σ_e	0,0159	0,0607	0,2753	-
80000	S_t	1,9718	3,8355	7,2721	69,9301
	S_e	1,9702	3,8462	3,3275	-
	σ_e	0,0042	0,0181	0,2328	-
160000	S_t	1,9759	3,8588	7,3706	81,9672
	S_e	1,9671	3,8527	3,4008	-
	σ_e	0,0051	0,0840	0,2610	-

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

Os resultados experimentais 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 como esperado pela análise do parágrafo anterior. 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, $0,9631$ e $0,9242$ 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**	0,8535	0,7673	0,2653
20000	1	$\mathbf{0,9631}$	0,8736	0,2970
40000	1	0,9705	$\mathbf{0,9242}$	0,3601
80000	1	0,9851	0,9615	0,4159
160000	1	0,9835	0,9631	0,4251

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

Uma pergunta que surge neste ponto é se a escalabilidade fraca 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 alcança valores satisfatórios (acima de 90% para $N > 30000$ e $p = 4$), 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 60% para $n < 10^5$ e $p = 4$). Assim podemos afirmar que para $k = 100$ o algoritmo possui escalabilidade fraca (apesar de lenta, pois a eficiência só se mantém acima 90% para $N > 30000$), mas para $k = 10$, o algoritmo já não é tão escalável como é para $k = 100$.

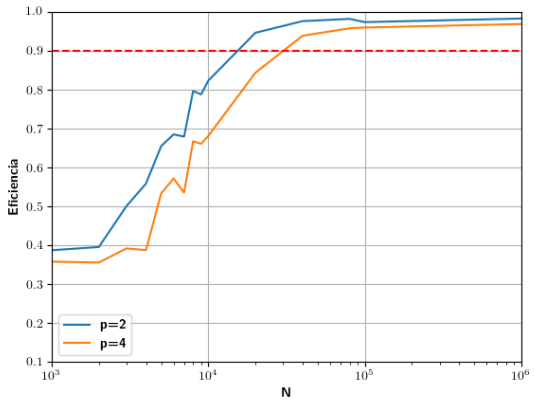


Figura 1: Eficiência como função de N para 2 e 4 processadores e $k = 100$.

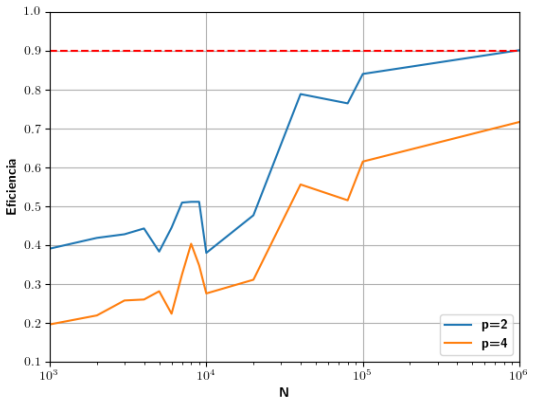


Figura 2: Eficiência como função de N para 2 e 4 processadores e $k = 10$.

A Tabela 4 mostra o *speedup* e a eficiência E para dados de tamanho da ordem de 10^6 obtidos do moodle, mostrando uma eficiência máxima de $96,01\%$ para $p = 2$ e $84,60\%$ para $p = 4$. O desvio padrão é mostrado entre parenteses.

N	p	2	4
1.10^6	$S_e(\sigma_e)$	1,7606 (0,0281)	2,6249 (0,0470)
	E	0,8803	0,6562
2.10^6	$S_e(\sigma_e)$	1,7932 (0,0132)	2,7405 (0,0101)
	E	0,8966	0,6851
5.10^6	$S_e(\sigma_e)$	1,9203 (0,0138)	3,3840 (0,0815)
	E	0,9601	0,8460

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

Finalmente a Figura 3 mostra o *overhead* do MPI e OpenMP (versão entregue no trabalho 1) como função de N, para $p = 4$ e $k = 10$. Percebe-se que para $N < 10^5$, o *overhead* é um pouco maior para MPI. Isto explica por que as curvas de crescimento da eficiência com N são ligeiramente mais lentas para o MPI quando comparadas com o OpenMP (vistas no trabalho 1). Por outro lado, para $N > 10^5$, os dois *overheads* ficam muito próximos indicando que o desempenho do OpenmMP e MPI são semelhantes para N grande. O motivo dos *overheads* serem parecidos é que estamos trabalhando em apenas uma máquina que possui memória compartilhada e o sistema operacional consegue trabalhar de forma inteligente e otimizada entre os processos.

A Figura 4 mostra o *overhead* dividido por N em função de N, ou seja, o *overhead* por ponto em função de N. Para ter escalabilidade, espera-se que o *overhead* tenha um crescimento mais lento que N, de forma que $overhead/N$ diminua com N. De fato observamos isso na figura 4, o $overhead/N$ diminui até estabilizar em um valor próximo de 2 ns. Sabendo que a frequência do processador é 3,6Ghz, obtemos um período de ciclo de relógio de $1/3,6\text{Ghz} = 0,27$ ns. Assim pode-se dizer que obtemos um *overhead* de $2\,ns/0,27\,ns = 7,2$ ciclos de relógio por ponto da entrada do algoritmo.

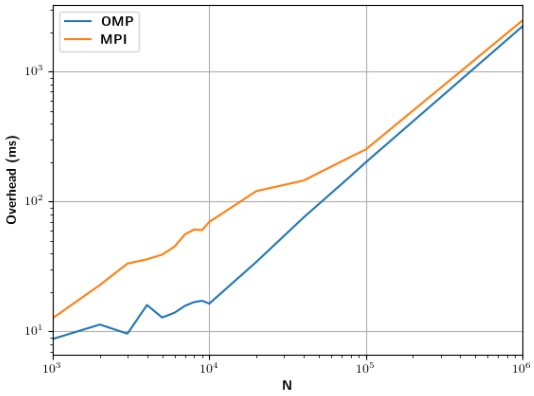


Figura 3: *Overhead* para OpenMP e MPI como função de N para $k = 10$ e $p = 4$.

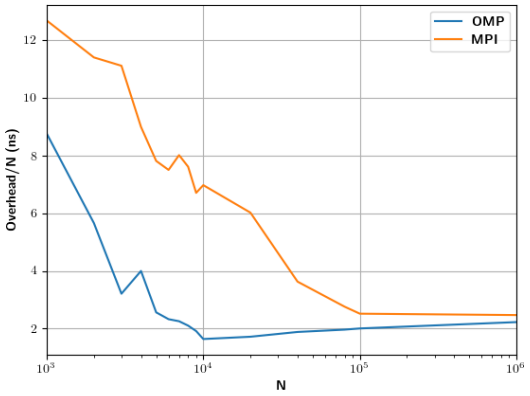


Figura 4: *Overhead*$/N$ para OpenMP e MPI como função de N para $k = 10$ e $p = 4$.

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.

Os testes de escalabilidade realizados indicam que o algoritmo não possui escalabilidade forte, pois a eficiência diminui consideravelmente quando dobramos o número de processadores mantendo N constante. Por outro lado, verificou-se que dependendo do valor de k, podemos ter escalabilidade fraca como visto para $k = 100$. Para $k = 10$, temos valores muitos baixos para a eficiência quando dobramos p e N. Para afirmar com certeza estes resultados, deve-se efetuar mais testes utilizando 8 e 16 processadores e verificar se a eficiência mantém estas taxas de decaimento.

Por último, verificou-se que o *overhead* com OpenMP e MPI são muito próximos. Assim pode-se dizer que quando estamos trabalhando com apenas uma máquina, estes dois métodos de paralelização possuem o mesmo desempenho, principalmente para N grande (maior que 10^5).