# Perfilagem do Algoritmo *Streamcluster*

Angelo Leite Medeiros Goes
20200000545
Jhonat Heberson Avelino de Souza
20200000680
Angelo Marcelino Cordeiro
20190152879
Maurício Thiago Ferreira de Lima
20180155222

## I. Descrição da Aplicação e Estrutura do Código

*Clustering* ou segmentação de dados é uma técnica de mineração de dados voltada ao agrupamento de dados segundo um conjunto de características, "*clusters*" [1]. A prática de "*clusterização*" tem como objetivo a análise exploratória e estatística de dados, sendo relevante a diversas áreas, como reconhecimento de padrões, aprendizagem de máquina, análise de imagens, compressão, entre outros.

Normalmente, algoritmos de *clustering* são realizados em bases de dados já coletadas e bem-definidas, entretanto, nem sempre os dados podem ou devem ser avaliados pós-geração, sendo necessário análises em tempo real como, por exemplo, gravações de telefone, dados multimídia, transações financeiras, entre outros. O algoritmo tratado neste relatório e nos projetos subsequentes trata-se de uma implementação de um algoritmo voltado a *clusterização* de dados em situação de *streaming*, em fluxo contínuo ou em tempo real.

O programa segue a estrutura de um algoritmo "k-means" [2] de agrupamento. Ou seja, dado um número K de centros, um arquivo com n-pontos d-dimensionais, definidos como argumentos iniciais, é possível encontrar a localização dos centros que melhor separam estes pontos [3].

Este processo é iterativo, e os centros são recalculados aleatoriamente até que a variância das distâncias centro-pontos não tenha grande alteração. O parâmetro de similaridade é justamente esse, da distância euclidiana. O trecho do código referente à esta implementação pode ser visto na listagem I.

```
/* compute Euclidean distance squared between two points */
float dist(Point p1, Point p2, int dim) {
    int i;
    float result = 0.0;
    for (i = 0; i < dim; i++)
    result += (p1.coord[i] - p2.coord[i]) * (p1.coord[i] - p2.coord[i]);
    return (result);
}
```

Listing 1. Trecho do Código

Argumentos passíveis de customização são: K-mínimo (k1) , K-máximo (k2), número de dimensões (d), número de pontos dos dados (n), número de pontos dos dados processados por iteração (*chunksize*), número máximo de pontos intermediários, arquivo de entrada (*infile*), arquivo de saída (*outfile*), e número de threads (*nproc*).

Sendo assim, o comando de execução da aplicação tem o formato descrito na listagem a seguir:

```
./streamcluster k1 k2 d n chunksize clustersize infile outfile nproc
```

Listing 2. Execução

Caso nenhum arquivo de entrada seja localizado e n $>$ 0, serão gerados n-pontos aleatórios no intervalo real (0, 1), para realização do agrupamento.

Para sincronização de threads são utilizadas *pthread barriers* [4], ou barreiras de threads, ao longo de toda execução do código.

## II. Descrição da metodologia de perfilamento

Após verificar que o código estava funcionando, o perfilamento foi realizado com o uso da ferramenta Gprof. O projeto foi então compilado, utilizando a variável de ambiente, através de 3 comandos com a flag "-pg", indicados na Listagem II.

```
/usr/bin/g++ -pg -O3 -g -funroll-loops -fprefetch-loop-arrays -fpermissive -fno-exceptions -
    static-libgcc -Wl,--hash-style=both,--as-needed -DPARSEC_VERSION=3.0-beta-20150206 -
    DENABLE_THREADS -pthread -c streamcluster.cpp

/usr/bin/g++ -pg -O3 -g -funroll-loops -fprefetch-loop-arrays -fpermissive -fno-exceptions -
    static-libgcc -Wl,--hash-style=both,--as-needed -DPARSEC_VERSION=3.0-beta-20150206 -
    DENABLE_THREADS -pthread -c parsec_barrier.cpp

/usr/bin/g++ -pg -O3 -g -funroll-loops -fprefetch-loop-arrays -fpermissive -fno-exceptions -
    static-libgcc -Wl,--hash-style=both,--as-needed -DPARSEC_VERSION=3.0-beta-20150206 -
    DENABLE_THREADS -pthread -L/usr/lib64 -L/usr/lib streamcluster.o parsec_barrier.o  -o
    streamcluster
```

Listing 3. Comandos

Inicialmente foram compilados os arquivos *streamcluster.o* e *parsec_barrier.o*, e por fim o binário executável *streamcluster*. A flag "-pg" foi usada para gerar o arquivo *gmon.out*, arquivo de análise Gprof.

Vários testes foram feitos, verificando como diferentes configurações afetam o fluxo do código, por exemplo, um teste de agrupamento de 10 pontos bidimensionais em dois grupos, mostrados na Tabela I. O resultado obtido instantaneamente está demonstrado na Tabela II

TABLE I
PONTOS USADOS PARA OS TESTES

| **x** | **y** |
|---|---|
| 0.041630 | 0.454492 |
| 0.834817 | 0.335986 |
| 0.565489 | 0.001767 |
| 0.187590 | 0.990434 |
| 0.750497 | 0.366274 |
| 0.351209 | 0.573345 |
| 0.132554 | 0.064166 |
| 0.950854 | 0.153560 |
| 0.584649 | 0.216588 |
| 0.806502 | 0.140473 |

TABLE II
PONTOS DOS CLUSTERS

| **Cluster** | Peso | **x** | **y** |
|---|---|---|---|
| C1 | 4 | 0.178246 | 0.520609 |
| C2 | 6 | 0.748801 | 0.202441 |

O resultado representa a localização dos dois centros, um com peso 4 e outro com peso 6. Ao realizar o mapeamento dos pontos, é possível observar que o resultado foi satisfatório. Dois grupos, um à esquerda e outro à direita foram formados, como podemos observar na Figura 1.

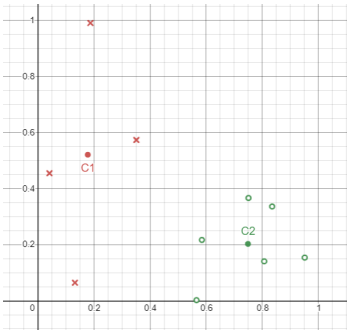


Fig. 1. Pontos mapeados

Também foram realizados testes para 100 pontos (0,01s), 1000 pontos (0,04s), 10000 pontos (0,8s) e 100000 pontos (15s). Há um crescimento exponencial do tempo, como é esperado.

## III. Resultados do perfilamento em uma CPU e associação dos gargalos com a estrutura do código

Após análise do perfilamento foi observado que a maior parte das chamadas e tempo de processamento estão distribuídas entre 5 funções específicas, duas delas tomando mais de 90% do tempo de execução para todas as configurações aferidas: "pgain" e "parsec_barrier_wait", como mostra a Listagem III. A função "pgain" é a função mais algoritmicamente exaustiva, já que nela é calculada o ganho na variância entre centros, para todos os pontos. Possui 2 "for" aninhados portanto sua complexidade é de O($n^2$), como mostra a Listagem III. Esta e suas auxiliares são as que mais geram gargalo.

```
./streamcluster 50 100 250 10000 1 10000 myin myout 2

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 82.98     16.75    16.75    24508     0.00     0.00  pgain(long, Points*, double, long*, int,
    parsec_barrier_t*)
 16.76     20.13     3.38   251412     0.00     0.00  parsec_barrier_wait(parsec_barrier_t*)
  0.30     20.19     0.06       14     0.00     0.00  pspeedy(Points*, float, long*, int,
    parsec_barrier_t*)
  0.00     20.19     0.00    19991     0.00     0.00  pkmedian(Points*, long, long, long*, int,
     parsec_barrier_t*)
...
```

Listing 4. Exemplo de Saída da Perfilagem (nproc igual a 2)

A função "parsec_barrier_wait" por sua vez implementa barreiras e para sincronização de threads [5]. Isso gera latência pois as threads ficam ociosas enquanto esperam umas às outras. Houve tempo perdido mesmo quando *nproc* = 1, ou seja, com somente uma thread, o que teoricamente não deveria acontecer já que não há o que sincronizar quando se tem apenas um processo. Por fim, a maior parte de suas chamadas vem de "pgain".

O tempo de execução aumenta com o número de pontos, mas não tanto com o aumento da dimensionalidade e número de agrupamentos. Foi observado que o tempo de execução cai drasticamente com o aumento dos *chunks* por iteração (*chunksize*), já que mais pontos eram lidos de uma vez, reduzindo a latência de leitura.

Também foi observado uma maior latência à medida que se aumenta o número de threads, devido especialmente ao aumento de tempo gasto nas barreiras de threads, e "for" que escalam com *nproc*.

```
double pgain(long x, Points* points, double z, long int* numcenters, int pid, pthread_barrier_t
    * barrier) {
...
    // at this time, we can calculate the cost of opening a center
    // at x; if it is negative, we'll go through with opening it

    for (int i = k1; i < k2; i++) {
        if (is_center[i]) {
            double low = z;
            //aggregate from all threads
            for (int p = 0; p < nproc; p++) {
                low += work_mem[center_table[i] + p * stride];
            }
            gl_lower[center_table[i]] = low;
            if (low > 0) {
                // i is a median, and
```

```
                // if we were to open x (which we still may not) we'd close i
                // note, we'll ignore the following quantity unless we do open x
                ++number_of_centers_to_close;
                cost_of_opening_x -= low;
            }
        }
    }
...
}
```

Listing 5. Aninhamento de For

## IV. CONCLUSÕES

Com o teste de perfilagem foi possível atestar algumas possíveis causas para um gargalo no agrupamento de números elevados de pontos, para uma CPU. Dentre eles: uso de *chunksize* diminuto, isso leva o núcleo a demorar muito somente na leitura dos pontos. Quando lê em pedaços maiores, reduz-se a latência.

Também, funções como "*pgain*" parecem gastar bastante tempo (aprox. 80%+), especialmente por ter maior parte dos *loops* no programa, ser a função mais algebricamente dispendiosa e ter complexidade O($n^2$). "*pgain*" também chama diversas funções auxiliares, como "parsec_barrier_wait", a segunda mais dispendiosa, responsável por sincronizar threads.

Se observou que com mais threads havia mais espera, logo mais atraso, que não deveria ser o caso já que o uso de mais processos deveria diminuir a latência. E mesmo com *nproc* = 1 (uma thread em execução) houve tempo perdido na função, comportamento julgado anômalo, já que não há o que sincronizar tendo um único processo. Tudo isso indicaria que há erro na implementação das barreiras.

## REFERÊNCIAS


[1] *Data Analysis 7: Clustering - Computerphile*. YouTube, Jul 2019. [Online]. Available: https://www.youtube.com/watch?v=KtRLF6rAkyo

[2] S. Learn, "k-means clustering," 2007. [Online]. Available: https://scikit-learn.org/stable/modules/clustering.html#k-means

[3] *StatQuest: K-means clustering*. YouTube, May 2018. [Online]. Available: https://www.youtube.com/watch?v=4b5d3muPQmA

[4] *Introduction to barriers (pthread_barrier)*. YouTube, Dec 2020. [Online]. Available: https://www.youtube.com/watch?v=_P-HYxHsVPc

[5] *Practical example to barriers (pthread_barrier)*. YouTube, Dec 2020. [Online]. Available: https://www.youtube.com/watch?v=MDgVJVIRBnM