# Trabalho Prático Nº 1

Computação Paralela, Mestrado Integrado em Engenharia Informática, Universidade do Minho

Júlio Beites Gonçalves
*Mestrado em Engenharia Informática*
*Universidade do Minho*
Braga
pg50537@alunos.uminho.pt

João Alexandre Gandra Ferreira
*Mestrado em Engenharia Informática*
*Universidade do Minho*
Braga
pg50461@alunos.uminho.pt

***Abstract*—Este trabalho tem como objetivo avaliar a aprendizagem das técnicas de otimização de código, através de análise de código e com recurso a ferramentas de análise de execução do mesmo.**

***Index Terms*—Optimização de código, C, vetorização, loop unroll, localidade espacial, localidade temporal**

## I. Introdução

Como base de suporte ao nosso processo de análise, desenvolvemos uma versão otimizada de um algoritmo k-means, sendo este baseado no algoritmo de Lloyd. As diretrizes obrigatórias de desenvolvimento do algoritmo são as seguintes apresentadas:

- Criar as N amostras e iniciar os K "clusters";
- a. Iniciar um vetor com valores aleatórios (N amostras no espaço (x,y));
- b. Iniciar os K clusters com as coordenadas das primeiras K amostras;
- c. Atribuir cada amostra ao cluster mais próximo usando a distância euclidiana;
- Calcular o centroide de cada "cluster" (também conhecido como centro geométrico);
- Atribuir cada amostra ao "cluster" mais próximo usando a distância euclidiana;
- Repetir os 2 passos anteriores até não existirem pontos que mudem de "cluster";

Como validação do resultado e output do programa os resultados do nosso programa devem corresponder os seguintes valores:

- N = 10000000, K = 4
- Center: (0.250, 0.750) : Size: 2499108
- Center: (0.250, 0.250) : Size: 2501256
- Center: (0.750, 0.250) : Size: 2499824
- Center: (0.750, 0.750) : Size: 2499812
- Iterations: 39

## II. Desenvolvimento do Algoritmo

### A. Primeira Fase

Inicialmente, desenvolvemos o algoritmo fazendo uso de duas estruturas de dados distintas como suporte ao nosso algoritmo.

```
typedef struct sPoint{
      float x;
      float y;
      int cluster;
} *POINT;

typedef struct sCluster{
     POINT centroid;
} *CLUSTER;
```

A primeira *sPoint*, serve para representar cada ponto através das suas coordenadas (x,y), associando ainda o cluster a que esta amostra pertencia. A segunda estrutura de dados *sCluster* permite associar a cada cluster o seu centroid. Com recurso a structs, conseguimos implementar o algoritmo. As primeiras métricas obtidas apontavam para valores na casa dos 12 segundos, sendo que as várias otimizações feitas ao código pouco impacto tinham no tempo de execução final. (Anexo 1)

### B. Segunda Fase

Após percebermos que a nossa estrutura anterior de suporte ao algoritmo apresentava grandes limitações a nivel de optimizações e tempo de execução, resolvemos alterar as mesmas recorrendo apenas a *arrays*.

```
float *pointX = malloc(N*sizeof(float));
float *pointY = malloc(N*sizeof(float));

int *pointCluster = malloc(N*sizeof(float));

float *clusterX = malloc(K*sizeof(float));
float *clusterY = malloc(K*sizeof(float));

int *numPoints = malloc(K * sizeof(int));
```

Os arrays *pointX*, *pointY* representam respetivamente as coordenadas X e Y de cada ponto, sendo cada ponto identificado pela sua posição no array.

Do mesmo modo, o array *pointCluster* serve para associar a cada ponto o cluster mais próximo.

Os arrays *clusterX*, *clusterY* representam respetivamente as coordenadas X e Y do centroid de cada cluster, sendo cada cluster identificado pela sua posição no array.

Por último, o array *numPoints* permite-nos quantificar o número de amostras associadas a cada cluster.

Com a implementação desta nova estrutura de dados de suporte ao algoritmo, ainda sem otimizações adicionais conseguimos decrementar o tempo de execução do programa para 8 segundos. (Anexo 2)

## III. Exploração e Otimizações do Código

Estando a base do nosso software agora otimizada, procedemos à exploração de possíveis incrementos de performance nas diversas operações executadas ao longo do código.

A primeira otimização efetuada ao nosso código base, centrou-se na nossa função de cálculo das distâncias euclidianas. Uma vez que o resultado desta função apenas era utilizado como forma de comparação com outras distâncias, nunca sendo o seu valor específico utilizado, o cálculo da raiz quadrada torna-se desnecessário. Do mesmo modo, verificamos que o cálculo da potência de expoente 2 neste caso, é uma operação com um custo superior ao da multiplicação, pelo que procedemos à mesma alteração.

Expressão Antiga:
*sqrt(pow(clusterX - pointX, 2) + pow(clusterY - pointY, 2));*

Nova Expressão:
*(clusterX - pointX)*(clusterX - pointX)) + ((clusterY - pointY)*(clusterY - pointY);*

Em todas, ou quase todas, as funções tentamos tirar o máximo partido da localidade espacial e da localidade temporal.

Na função *calculateCentroid* que calcula o centróide de um *Cluster* há a utilização da localidade espacial em ambos os ciclos *for*, além da localidade temporal que existe no array *xSumCluster* e *ySumCluster*. Outro exemplo é na função *bestCluster* que dado um ponto e todos os centróides calcula o melhor *Cluster* para o ponto dado, que usa a localidade espacial nos arrays *clusterX* e *clusterY* além de utilizar localidade temporal nas variáveis *pointX* e *pointY*.

Na função *bestCluster*, utilizávamos uma verificação a cada iteração do ciclo *for* para verificar se a distância calculada era menor do que a menor distância até aí. Utilizávamos também um número de iterações que não era múltiplo do grau de unrolling, pois a primeira iteração era efetuada fora do ciclo para termos base para comparação, o que não facilitava o loop unrolling nem a vetorização. Por esse motivo pareceu-nos razoável transformar nos 2 ciclos *for* atuais, passando todas as branches (if) para outro ciclo o que facilita o loop unrolling e a vetorização do primeiro. Adicionamos também a diretiva *pragma omp simd*, que é uma diretiva que permite identificar os ciclos que permitam que as suas iterações sejam efetuadas de forma concorrente.

Outra otimização que fizemos foi na contagem do número de pontos em cada cluster. Na primeira versão do nosso algoritmo, os pontos eram recontados a cada iteração, sendo que agora só no caso de uma mudança de cluster é que há uma retificação nessa variável.

## IV. Conclusão e melhorias possíveis

Apesar de acharmos que o nosso algoritmo ficou com uma otimização boa, consideramos que poderíamos ter ido mais além. A nossa função *calculateBestCluster* apresenta um número elevado de branches (if) no ciclo *for* o que prejudica a vetorização e o loop unrolling, pelo que achamos que deveria ser reformulado para obter uma otimização ainda melhor.

Analisando os valores finais depois da otimização (Anexo 3) e os valores iniciais (Anexo 1), verificamos que houve uma diminuição de 70% nas misses da cache, uma diminuição de 50% nas instruções efetuadas e uma diminuição de 75% nos ciclos de processador necessários. O CPI também desceu para metade, o que indica que as instruções, além de terem diminuído abruptamente, também diminuíram de complexidade.

# V. Attachments

```
Performance counter stats for './bin/k_means' (5 runs):

       526121386      L1-dcache-load-misses                                       ( +-  0,30% )
     61492523139      inst_retired.any          #      0,8 CPI                    ( +-  0,07% )
     46749380679      cycles                                                      ( +- 10,67% )

           15,39 +- 1,91 seconds time elapsed  ( +- 12,38% )
```

Fig. 1. Anexo 1

```
Performance counter stats for './bin/k_means' (5 runs):

       167093915      L1-dcache-load-misses                                       ( +-  0,04% )
     48799750642      inst_retired.any          #      0,6 CPI                    ( +-  0,00% )
     27852552314      cycles                                                      ( +-  0,03% )

          8,7464 +- 0,0986 seconds time elapsed  ( +-  1,13% )
```

Fig. 2. Anexo 2

```
Performance counter stats for './bin/k_means' (5 runs):

       168686484      L1-dcache-load-misses                                       ( +-  0,05% )
     32031199203      inst_retired.any          #      0,4 CPI                    ( +-  0,00% )
     12305357909      cycles                                                      ( +-  0,02% )

          3,7885 +- 0,0580 seconds time elapsed  ( +-  1,53% )
```

Fig. 3. Anexo 3