Computação Paralela
Algoritmo K-Means
Enunciado Prático 2

José Diogo Vieira
Departamento de Informática
Universidade do Minho
Braga,Portugal
pg50518

António Luís Fernandes
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg50229

***Abstract*—Dado um array inicial com valores aleatórios (x,y)), irá inicializar os K centroids com os primeiros K valores. Depois disso, vai associar cada ponto ao cluster que se encontrar mais próximo através da distância euclidiana. Por fim, depois de todos os pontos atribuídos, são calculados os novos valores dos centroids e repete-se o processo até não haver nenhum ponto a trocar de centroid. Neste relatório, iremos explicar a transição da implementação deste algoritmo sequencial para paralelo com recurso às primitivas do OpenMP.**

***Index Terms*—computação paralela, k-means, paralelismo, threads, openMP, otimização**

I. Introdução

Neste segundo trabalho prático e dando continuidade ao projeto desenvolvido anteriormente, o principal objetivo foi avaliar os benefícios em termos de tempo de execução ao identificar os blocos de maior carga computacional e utilizar técnicas de paralelismo para melhorar o desempenho.

II. Alterações da versão sequencial

Antes de explorar a nossa implementação do paralelismo, foram feitas algumas alterações relativamente à implementação sequencial. Foi alterado o critério de paragem, não sendo necessário agora a verificação se algum ponto mudou de cluster. Fixando o critério de paragem nas 20 iterações.

Foi necessário remover os defines para o número de samples e clusters, e torná-los variáveis globais que serão inicializadas no começo do programa visto que este valores serão passadas como argumentos, isto vai implicar que certos arrays que seriam inicializados em compile time passarem a serem em run time.

III. K-Means Paralelo

A. Identificação dos blocos de código

De modo a identificar quais os blocos de código com maior carga computacional, utilizamos o comando: ”perf record” e ”perf report”, de modo a obter essa informação. Repetimos esta etapa 2 vezes, uma para 4 clusters e outra para 32, para ver como é que a carga variava com o aumento do número de clusters. Verificámos que a maior parte da carga computacional do programa se encontrava na função que indica a que cluster pertence cada ponto, como já era de esperar. E com o aumento do número de clusters, esta carga torna-se cada vez maior.

Em segundo lugar, temos o cálculo dos centroids que inclui a soma dos valores de cada centroid e a sua respetiva divisão pelo tamanho de cada cluster. À medida que o número de cluster aumenta, esta tem um menor valor de carga comparando com a anterior.

Serão por isso, estes os blocos de código que vamos incidir mais para o paralelismo.

B. Exploração do paralelismo

Para o bloco de códido da atribuição de um cluster a cada ponto, a primeira intuição foi paralelizar o loop exterior, podendo assim se efetuar mais iterações em paralelo sem nenhuma preocupação, visto que não há partilha de dados entre as threads.

Adicionamos assim as seguintes diretivas do **OpenMP**, imediatamente antes do for exterior:

```
#pragma omp parallel for num_threads(NumThreads)
```

Desta forma, o loop exterior seria paralelizado e cada iteração deste seria feita sequencialmente. Se quisessemos paralelizar o loop interior irá ser mais complexo, pois este, tem dependência entre iterações e como já estamos a parelizar o ciclo mais exterior não podemos utilizar outra vez o paralel for.

Podemos tentar usar a diretiva collapse, que vai dividir as threads por N*K, tendo em consideração que esta estratégia faria mais sentido se o número de N fosse menor do que o K.

Mas para conseguir utilizar esta estratégia será necessário fazer algumas alterações.

```
lowest = 10
#pragma omp parallel for \
          num_threads(NumThreads) \
          collapse(2) firstprivate(lowest))
for i = 0 to N
```

```
for k = 0 to K
  dist = distance(vector[i],centroid[k])
  if (dist < lowest)
    lowest = dist
    index_low = k
    if (k==K-1) lowest=10
```

Esta versão adiciona uma condição ao ciclo interior, para o lowest ser inicializado no final de cada iteração, esta condição adiciona mais complexidade e número de instruções ao programa, o que prejudica o tempo da execução do programa, descartamos assim esta versão.

Tentamos também explorar uma versão em que utilizávamos a diretiva *reduce* para o mínimo, mas não conseguimos fazer com que funcionasse corretamente.

Escolhemos assim, a primeira versão apresentada para a versão final.

Passamos assim para a segunda função com maior carga computacional, o cálculo de centroids.

Esta função é dividida em duas partes:

A primeira parte que consiste em somar os valores dos clusters e o tamanho destes.

```
for i = 0 to N
    index_low = cluster[i]
    sumX[index_low] += vectorX[i]
    sumY[index_low] += vectorY[i]
    tSize[index_low]++
```

Ao utilizarmos a parelelização com o for, temos de ter atenção que existem dependências entre iterações, visto que uma dada posição dos três arrays (sumX,sumY e tempSize) será incrementada. Para combater estas dependências temos algumas opções como a diretiva $critical$, $atomic$ ou $reduce$. Ao exprimentar as três, verificamos que a melhor para usar seria a diretiva $reduce$. Assim, iremos colocar o seguinte imedietamente antes do for:

```
pragma omp parallel \
    for num_threads(NumThreads) \
reduction(+:sumX[:K],sumY[:K],tSize[:K])
```

Para a segunda parte, que consiste no cálculo dos centroids, será usada apenas as diretivas parallel e for.

```
#pragma omp parallel for \
    num_threads(NumThreads)
for i = 0 to K
    size[i] = tSize[i];
    centroid[X(i)] = sumX[i]/tSize[i];
    centroid[Y(i)] = sumY[i]/tSize[i];
```

IV. Desempenho da proposta implementada

A. Testes realizados

Foram realizados testes para avaliar o desempenho da proposta implementada. Estes testes foram realizados no Cluster Search na partition cpar, com recurso ao comando perf. Todos os resultados são uma média de 5 execuções. Serão apresentados os resultados do tempo de execução juntamente com o valor de *speed-up* (que relaciona o tempo da execução sequencial com o tempo da execução paralela) para 4 e 32 clusters, respetivamente:

NºThreads	Tempo de Execução(s)	Speed-up
1 (Sequencial)	2.3143	1
2	2.1382	1.082
4	1.2001	1.928
8	0.7493	3.089
16	0.4807	4.815
32	0.3728	6.208
40	0.3512	6.569

TABLE I: Resultados para 4 clusters

NºThreads	Tempo de Execução(s)	Speed-up
1 (Sequencial)	14.1611	1
2	14.0397	1.009
4	6.8222	2.076
8	4.1790	3.389
16	2.5180	5.624
32	1.3262	10.678
40	0.9753	14.520

TABLE II: Resultados para 32 clusters

B. Análise dos Resultados

Com estes resultados, conseguimos observar o aumento da performance (menor tempo de execução) com o aumento do nº de $threads$ tanto para 4 clusters como para 32 (Fig 1 e Fig 2).

Verificamos um grande aumento de tempo de execução par a versão sequencial com o aumento do número de clusters, o tempo aumenta 7x. Em contraste com a versão em paralelo em que os tempos são muito menores em comparação com a sequencial e têm um aumento entre 3-4x, observa-se assim que em termos de escabilidade para o número de clusters, a versão paralela escala muito melhor. Dentro da versão paralela, com o aumento do nº de threads maior a escalabilidade.

Em termos de balanceamento de carga, o bloco de código com a atribuição de um cluster a cada ponto mantém-se predominante nos dois testes, e a sua carga aumenta consoante o aumento de clusters. Com o aumento de threads, observa-se também um aumento desta carga, mas este efeito é causado pela diminuição da carga no cálculo dos centroids.

V. Conclusão

Em suma, conseguimos identificar e explorar o paralelismo nos blocos de código com maior carga computacional, testando o seu desempenho e fazendo uma análise desses resultados.

A nossa maior dificuldade passou pela paralelização do bloco da atribuição de um cluster a cada ponto, tentamos várias alternativas, mas acabamos por ficar versão em que cada iteração será sequencial, achamos que poderá haver uma melhor alternativa do que esta, mas não conseguimos descobrir qual.

Apesar disso, passamos a compreender melhor a utilização das diversas primitivas do OpenMP tal como a diferença da metodologia no desenvolvimento de programas paralelos e sequencias.

APPENDIX

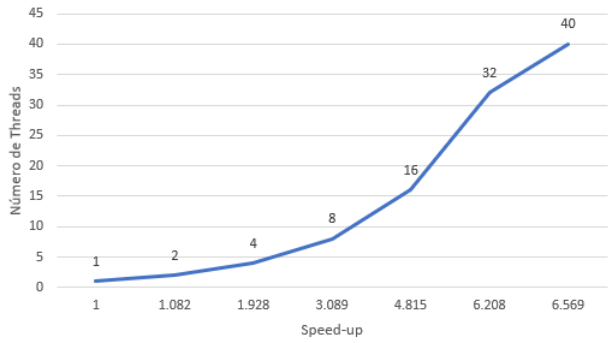

Fig. 1: Speed-up 4 clusters

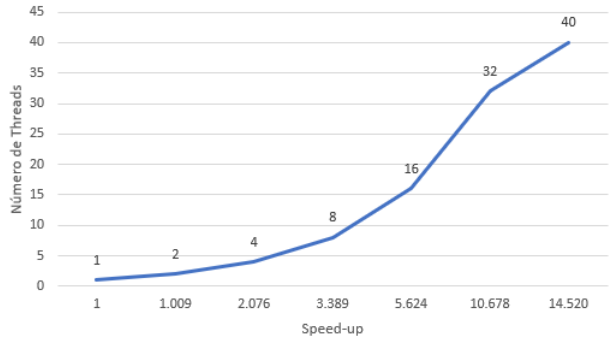

Fig. 2: Speed-up 32 clusters