Computação Paralela
Algoritmo k-Means

1st Catarina Martins
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50289@alunos.uminho.pt

2nd Joaquim Roque
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50502@alunos.uminho.pt

***Resumo*—O algoritmo *k-Means* é um método de agrupamento de pontos que subdivide um conjunto de N amostras em k *clusters*. É um processo iterativo em que a cada amostra é-lhe atribuído um *cluster*, com base na menor distância euclidiana ao centro do mesmo (referido como centroide). O processo iterativo termina quando se verifica que o algoritmo convergiu, ou seja, nenhum ponto mudou de *cluster*.**

Index Terms*—otimização, *performance*, vetorização, paralelismo, centroides, *cluster

I. Introdução

O objetivo deste trabalho consiste na implementação e otimização do algoritmo em questão, com recurso às ferramentas usadas nas aulas práticas. O programa deve ser escrito em C e a implementação do mesmo deve ser sequencial, recorrendo a diversos meios de otimização, desde o uso de *flags* durante a compilação (i.e. *-O2* e *-O3*), a alterações ao próprio algoritmo.

II. Implementação

A. Versão I

Numa versão inicial, desenvolvemos uma API para representação, manipulação e consulta das estruturas correspondentes às amostras e aos vetores. Cada *cluster* é representado pelo seu centroide e por um vetor de amostras, correspondente aos pontos pertencentes ao mesmo no momento.

O algoritmo inicia-se com a alocação das estruturas supra mencionadas, e inicialização das mesmas com valores aleatórios (i.e. $(x, y) \in [0, 1]^2$). Os quatro primeiros pontos são atribuídos a cada *cluster*, tomando o papel de centroides inciais. Seguidamente, inicia-se o processo iterativo de cálculo de distâncias euclidianas, com objetivo de computar o índice do *cluster* ótimo, ou seja, aquele cuja distância ao ponto em questão é menor. Daí é guardada uma **cópia** da amostra no respetivo *cluster*, para posteriormente ser calculado o novo centroide. Através de um booleano é possível verificar se o algoritmo convergiu ou não, tendo por base o facto de haver pelo menos um ponto a trocar de agrupamento.

Esta versão é particularmente ineficiente, uma vez que em cada iteração são feitas N cópias das estruturas das amostras. Para além disso, notamos que determinar o tamanho dos vetores internos aos *clusters* era especialmente complexo, devido à imprevisibilidade nas atribuições aos *clusters*. Tal implicava o uso de realocações das estruturas em memória. Finalmente, apesar de aparentemente o vetor das amostras ser percorrido uma única vez por iteração, o uso de vetores auxiliares faz com que, na realidade, seja percorrido duas vezes.

B. Versão II

Na segunda versão, procedemos à alteração das estruturas, de modo a que a alocação de vetores auxiliares internos aos *clusters* não fosse necessária. Para tal, cada amostra é marcada com uma *tag*, que não é mais que o índice do *cluster* ao qual está atribuída. No entanto, a estrutura que representa cada agrupamento passa, agora, a dispor de uma variável que indica o número de amostras que pertencem ao mesmo. Outra otimização a destacar em relação à versão anterior é a redução efetiva para metade do número de iterações sobre o vetor de amostras. Para tal, a soma das coordenadas dos pontos é feita no mesmo ciclo em que é calculado o *cluster* ótimo.

Nos ciclos de cálculo das distâncias euclidianas, procuramos eliminar o uso de *if-clauses*, substituindo-as por operadores ternários, uma vez que estas impedem a auto-vetorização por parte do compilador. Contrariamente ao esperado, esta alteração não teve impacto na vetorização, uma vez que as variáveis exibem dependências do tipo *Read after Write*. Esta informação pode ser obtida do *gcc* usando a *flag -fopt-info*. De notar ainda que todos os apontadores nos parâmetros das várias funções foram marcados com a *keyword restrict*, mas, mais uma vez, esta alteração não teve impactos significativos.

C. Versão III

Na versão final, optamos por condensar o algoritmo e as estruturas necessárias num único ficheiro de código fonte, isto para ser possível ao compilador reduzir o número de chamadas a funções (*inlining*). Tal não era possível nas versões anteriores devido ao encapsulamento das estruturas nas respetivas unidades de tradução.

III. Resultados

Todas os executáveis foram compilados com *gcc -O3 -DNDEBUG* (versão 7.2.0), e posteriormente corridos com o comando *srun –partition=cpar perf -e instructions,cycles -M cpi bin/k_means*. O comando foi executado 5 vezes em cada

caso e registada a melhor dessas iterações para a análise de resultados que se segue.

Tabela I
RESULTADOS VERSÃO I

Métrica	Valor
CPI	0,8
Ciclos	33.000.000.000
#I	40.600.000.000
Tempo(s)	11,0

Tabela II
RESULTADOS VERSÃO II

Métrica	Valor
CPI	1,0
Ciclos	29.000.000.000
#I	30.000.000.000
Tempo(s)	9,0

Tabela III
RESULTADOS VERSÃO III

Métrica	Valor
CPI	0,4
Ciclos	13.700.000.000
#I	32.300.000.000
Tempo(s)	4,0

IV. ANÁLISE DE RESULTADOS

Como se pode verificar o número de ciclos foi baixando a cada versão bem como o tempo de execução. Um baixo CPI justifica-se devido ao aumento do paralelismo, sobretudo visível na versão III. No entanto, há que destacar o facto de que o número de instruções é consideravelmente superior na versão III em relação à versão II, e, apesar disso, o tempo de execução é menos de metade. Dessa forma, conclui-se que estas métricas de análise de *performance* não devem ser analisadas de forma isolada como meio de comparação de implementações do mesmo programa/algoritmo.

V. CONCLUSÃO

Com a realização deste trabalho, pudemos adquirir alguma sensibilização relativamente a questões de paralelismo, análise de *performance* e otimização. Como aspetos negativos, destacamos o facto de não termos conseguido obter auto-vetorização neste algoritmo, particularmente nas secções do código mais exigentes computacionalmente. No entanto, há a destacar pela positiva o facto de termos conseguido atingir uma *performance* bastante razoável face ao que foi estabelecido como objetivo por parte da equipa docente.