Trabalho Prático 1

Computação Paralela

1st Duarte Augusto Rodrigues Lucas - PG50345
Universidade do Minho
Mestrado em Engenharia Informática
Santo Tirso, Portugal
pg50345@alunos.uminho.pt

2nd Tiago Fernandes Ribeiro - PG50779
Universidade do Minho
Mestrado em Engenharia Informática
Barcelos, Portugal
pg50779@alunos.uminho.pt

***Abstract*—Este documento é relativo ao trabalho prático 1 da unidade curricular de Computação Paralela, que tem como objetivo o desenvolvimento, análise e otimização de um algoritmo k-means simples, baseado no algoritmo de Lloyd.**

***Index Terms*—computação paralela, otimização, análise, linguagem C, algoritmo k-means**

I. Introdução

Neste projeto foi desenvolvido em C uma versão sequencial otimizada de um algoritmo k-means simples, com base no algoritmo de Lloyd. Este algoritmo classifica um conjunto de amostras (pontos) em k grupos (clusters).

Mantendo sempre a filosofia do algoritmo original, o objetivo passa por minimizar o tempo total de execução do algoritmo, utilizando técnicas de otimização de código, recorrendo a técnicas e ferramentas de análise de código.

II. Desenvolvimento do código

A. Descrição algoritmo k-means (Lloyd)

1) Criar as amostras e iniciar os "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
2) Calcular o centroide de cada "cluster" (também conhecido como centro geométrico)
3) Atribuir cada amostra ao "cluster" mais próximo usando a distância euclidiana
4) Repetir os passos 2 e 3 até não existirem pontos que mudem de "cluster"

B. Implementação

Inicialmente foi feita a implementação básica do algoritmo k-means sem pensar no tempo de execução. Esta implementação passou por etapas de extrema importância, tal como a inicialização dos pontos e dos clusters, a atribuição das amostras aos vários clusters e de seguida o início das várias iterações onde era recalculado o centroide de cada cluster e eram atribuídas de novo as amostras aos clusters, tendo em atenção que se nenhuma amostra alterar de cluster, significa que o algoritmo convergiu e foi encontrada a solução.

Para a representação de cada ponto foi construída uma struct com dois valores *float* relativos às coordenadas do ponto e um *int* que indica o cluster em que está inserido. Em contrapartida, para a representação do cluster também foi utilizada uma estrutura de dados com um ponto, que representa o centroide, um *int* que indica quantos pontos tem o cluster e dois *floats* que apontam o somatório das coordenadas dos pontos do cluster, que têm como propósito auxiliar o cálculo do centroide da próxima iteração.

Inicialmente, através do método *inicializa*, são atribuídas coordenadas (x,y) aleatórias ao pontos, código que foi inicialmente disponibilizado pelos docentes da disciplina, recorrendo ao método *srand*. A equipa de trabalho posteriormente deu continuidade ao método criando um array com N pontos com o método supramencionado. Foi também desenvolvido um array de clusters em que os seus centroides foram preenchidos com os primeiros K pontos gerados.

A atribuição das amostras aos clusters é feita no método *atribuiCluster*, onde é percorrida a lista de pontos, sendo que para cada ponto é iterada a lista de clusters para determinar a menor distância euclidiana entre o ponto atual e o centroide de cada cluster. O ponto é adicionado ao cluster onde a distância entre o ponto e o seu centroide é menor, sendo que depois é atualizado o tamanho do cluster e o somatório das coordenadas dos pontos.

Depois da inicialização dos clusters é realizado o cálculo dos novos centroides, no método *calculaCentroid*, onde são percorridos todos os clusters e é dividido o somatório das coordenadas dos pontos do cluster pelo tamanho do mesmo, para calcular o centro geométrico.

Com os novos centroides é realizada uma nova atribuição das amostras pelos clusters, isto até que não exista mais nenhum ponto que altere de cluster. Ao percorrer todos os pontos é atualizada uma *flag* que irá mudar quando algum ponto ficar num cluster diferente da iteração anterior. Se esta *flag* não se alterar, concluímos que o algoritmo convergiu.

III. Análise

No decorrer da implementação do algoritmo, foram testadas várias soluções que acabariam por chegar ao mesmo resultado,

porém de maneira menos eficiente, sendo desta forma descartadas. Foi então pretendido aproveitar ao máximo todos os ciclos realizados, de modo a não obter redundâncias onde seria necessário realizar iterações adicionais sobre os arrays, sendo que estas podiam ser embutidas dentro de ciclos já executados. Por exemplo, enquanto são atribuídas as amostras ao clusters, é realizado o cálculo do somatório das coordenadas dos pontos que será usado para calcular os próximos centroides.

Apesar do cálculo da distância euclidiana ser possível recorrendo a métodos existentes, estes devem ser substituídos por instruções de menor custo, pois em maior parte dos compiladores estas funções geram um código *assembly* com instruções mais poderosas que aumentam o tempo de execução do programa.

Relativamente a localidade espacial esta é evidenciada quando são efetuadas iterações sobre todos os pontos, é possível transferir informação sobre vários pontos da memória para a cache, sendo que o mesmo acontece quando os clusters são percorridos. Quanto à localidade temporal, esta acontece quando iteramos sobre os clusters para descobrir a menor distância euclidiana entre o ponto e o centroide, onde é acedida a informação sobre o mesmo ponto num curto espaço de tempo para realizar os cálculos.

IV. Otimização

Para otimizar o cálculo da distância euclidiana não foram utilizados métodos como o *pow* e *sqrt*. Como apenas é feita uma comparação entre duas distâncias a raiz quadrada é prescindível, se $\sqrt{x} > \sqrt{y}$ então $x > y$. Quanto ao cálculo do quadrado da expressão, foi utilizado *inlinning*. Multiplicar a expressão por ela mesma é mais eficiente que utilizar o método *pow*, pois produz instruções mais eficientes e rápidas.

Quanto às otimizações relativas ao compilador, foram realizados vários testes com as várias opções que o compilador *gcc* tem para oferecer, para comparar e concluir qual seria o melhor conjunto de otimizações. Mesmo antes de executar o programa, é possível concluir que apenas utilizando o nível -O2 de otimização será muito benéfico para diminuir o tempo de execução, pois como já foi referido, o facto de existir localidade espacial e temporal irá fazer com seja possível fazer uso da vetorização, algo que é muito importante para obter melhores tempos de execução. A maneira como as estruturas de dados estão armazenadas na memória faz com que seja possível o uso desta otimização.

V. Resultados

Para uma boa interpretação dos resultados obtidos o programa desenvolvido sofreu uma série de testes na máquina virtual disponibilizada pelos docentes da unidade curricular, o cluster SeARCH. O programa foi compilado com diferentes níveis de otimização, de modo a obter os vários tempos de execução. Para tornar os resultados mais coesos, o programa foi executado 5x em cada nível de otimização.

	#1	#2	#3	#4	#5	Média
-O2 - Time	**6.32**	**9.14**	**5.87**	**6.08**	**5.91**	**6,66**
-O2 - CPI	**0.56**	**0.83**	**0.54**	**0.54**	**0.54**	**0,60**
-O3 - Time	**7.33**	**9,93**	**7.62**	**9.20**	**10.31**	**8,88**
-O3 - CPI	**0.94**	**1,25**	**0.95**	**1.15**	**1.32**	**1.12**

TABLE I
TEMPO DE EXECUÇÃO NO CLUSTER SEARCH

Com estes resultados, vemos que o nosso programa beneficia mais do nível de otimização -O2. Como supramencionado na análise realizada ao código, foi previsto que a vetorização iria reduzir bastante o tempo de execução. A vetorização permite a realização de mais do que um calculo matemático ao mesmo tempo, transferindo vários valores da memória para a cache de uma só vez. A *flag* -O2 executa quase todas as otimizações suportadas que não envolvem perdas significativas tanto em tempo de execução como em espaço utilizado na memória. Em contrapartida, o nível de otimização -O3 não atinge tempos tão bons, logo o programa não beneficia tanto de conceitos como o *loop unrolling* e a troca de *loops* quando estes estão aninhados. Apesar deste nivel de otimização ter todas as otimizações dos níveis anteriores, este tem um desempenho inferior quando comparado com o conjunto de otimizações -O2.

VI. Conclusão

A unidade curricular de Computação Paralela tem como objetivo apresentar-nos conceitos que nos levam a melhorar ideias como a otimização e análise do código. Com a realização deste trabalho foi possível consolidar toda a matéria lecionada nas aulas teóricas e práticas, uma vez que nos permitiu estudar e praticar os temas anteriormente referidos manipulando a ordem com o que código era executado, as *flags* usadas e métodos recomendados pelos docentes da disciplina.

Em suma, concluímos que este trabalho prático constitui uma mais valia para o nosso conhecimento, visto que permitiu adquirir boas bases para trabalhos futuros, quer académicos ou profissionais. Consideramos também que em relação ao pretendido pelos docentes da cadeira para este primeiro trabalho prático, conseguimos uma boa resolução do problema, tendo em conta o tempo de execução final do programa.

References

[1] datasciencelab, "Clustering With K-Means in Python," https://datasciencelab.wordpress.com/tag/lloyds-algorithm/