Trabalho Prático 2 - Paralelização de Algoritmo K-Means simples baseado no algoritmo de Lloyds

1st Eduardo Teixeira[pg47166]
Escola de Engenharia
Universidade do Minho)
Braga, Portugal
pg47166@alunos.uminho.pt

2nd Marco Sampaio[pg47447]
Escola de Engenharia
Universidade do Minho)
Braga, Portugal
pg47447@alunos.uminho.pt

***Abstract*—O seguinte trabalho foca-se na análise de paralelismo e na avaliação do desempenho deste tipo de programação num algoritmo que trata da segregação de N dados em torno k clusters - algoritmo k-means.**

***Index Terms*—Lloyd, k-means, Otimização, Performance, Análise, C, Paralelismo, *OpenMP*, Exclusão Mútua, Sincronização, Threads**

I. Introdução

O objetivo deste segundo trabalho prático vai de encontro à avaliação da capacidade dos alunos para desenvolver programas que explorem paralelismo tendo como principal objetivo a redução do tempo execução deste.

De forma a realizar este trabalho, foi necessário aprimorar um algoritmo simples de K-Means baseado no algoritmo de Lloyd's focando na análise computacional e na escolha dos blocos de código mais apropriados à implementação de paralelismo, sempre com o objetivo de minimizar o tempo total de execução do nosso algoritmo. Neste contexto é utilizada a API estudada na cadeira o *openMP*, onde adicionamos ao código já feito primitivas de paralelismo, de sincronização e de atomicidade.

II. Alterações para a fase atual

O grupo apercebeu-se que com as estruturas de dados implementadas no trabalho passado a paralelização do algoritmo se tornava bastante difícil e complicada de conseguir resultados satisfatórios. Deste modo foram trocadas as structs *Amostra* e *Cluster* por arrays como se pode ver na seguinte lista:

- **float *amostrasx** : coordenadas x das amostras
- **float *amostrasy** :coordenadas y das amostras
- **float *clustersx** : coordenadas x dos clusters
- **float *clustersy** : coordenadas y dos clusters
- **int *clusters** : centroid mais próximo de cada amostra
- **float *somax** : soma das coordenadas x de cada cluster
- **float *somay** : soma das coordenadas y de cada cluster
- **int *total** : total de amostras de cada cluster

Esta alteração comprovou ser bastante importante pois o tempo de execução do algoritmo foi melhorado consideravelmente. Ao modificar-se as estruturas de dados e consoante o enunciado, o grupo apercebeu-se de que o modo de utilização da variável **continua** não iria ser importante para o algoritmo, pois este nunca iria convergir e porque foi estabelecido um critério de paragem nas 20 iterações. Devido a esta alteração a região crítica do algoritmo foi alterada. Foi adicionado um novo ciclo for, onde para cada ponto/amostra se identifica o centroid mais próximo, se adiciona as coordenadas de cada ponto (arrays somax e somay) e se incrementa o total de pontos/amostras de cada cluster (array clusters). A região crítica passa então a pertencer a este "novo" for.

III. Análise de carga Computacional do Algoritmo

Na análise de carga computacional do algoritmo foi, primeiramente analisado o output de assembly associado ao executável k_means onde podemos dividir os blocos pelo número de ciclos for utilizados. Foi então ao longo da análise do ficheiro que se determinou o peso da inicialização das amostras e dos clusters, tendo este bloco/for pouco carga computacional.

De seguida focamos-nos na função ***findclosestCentroids()***, nesta função temos realmente os blocos de código que possuem carga computacional. Podemos dividir esta função em 4 blocos: o primeiro, que inicializa os clusters no seu *size* e nas respetivas coordenadas; o segundo, onde se calcula e verifica para cada ponto a distância e o novo centroid;em terceiro o "novo" for onde se iteram N pontos para não só adicionar as coordenadas aos arrays de somatórios (arrays somax e somay) como para incrementar o total de pontos de cada cluster ; e em 4, o bloco onde atribui um novo valor de coordenadas a cada centroid. Como k não é um valor necessariamente grande então o primeiro bloco possui uma carga computacional baixa. Algo que não acontece no segundo bloco onde se realmente fazem os cálculos e se atribui o centroid mais próximo a cada amostra, estas operações possuem grande parte do peso computacional do nosso algoritmo. No terceiro bloco, voltamos a ter bastante peso computacional pois iteramos sobre o número total de pontos, é também neste bloco que identificamos a região crítica do nosso algoritmo, deste modo este bloco é considerado importante na execução do algoritmo . O quarto bloco relaciona-se com o primeiro visto que o número de iterações é baixo, de acordo com o valor de k, e desta forma a carga computacional é baixa. Através do output *perfreport* do de assembly gerado pela ferramenta *perf* comprovamos a análise feita.

IV. Alternativas para exploração de paralelismo para cada bloco identificado em II

Uma vez que no bloco identificado em II possuímos ciclos for com alta probabilidade de serem elegíveis para paralelizar o código. É neste âmbito que são então exploradas diferentes *contructs* e *directives* do *openMP* para a paralelização. Como é num ciclo for que constitui grande parte do peso computacional, foi utilizada a cláusula ***pragma omp parallel for*** que paraleliza pelo número de threads escolhidas o bloco de código que compõem o for. Desta diretiva nasceu a dúvida de se estávamos num caso específico onde um *scheduling* de threads particular como *schedule(dynamic)* faria sentido para o ciclo. Porém verificou-se que esta opção não era certo visto que a*dynamic* trabalhava bem quando o peso de computação das instruções varia a cada iteração, deste modo optou-se pela divisão das iterações pelas threads pelo *schedule(static)*. Neste segundo bloco possuimos paralelização, o comportamento default no final do for é existir sincronização, ou seja, utiliza-se a cláusula ***pragma omp barrier*** onde que as threads numa região paralela esperam até que todas acabem a sua execução . Visto que a região crítica não se encontrava neste

ciclo o grupo não efetuou mais alterações às cláusulas deste segundo bloco.

No terceiro bloco analisado em II, o grupo procedeu à paralelização do ciclo com ***pragma omp parallel for***, porém ao paralelizarmos a região seria necessário ter cuidado com o a sincronização de threads, pois estávamos a aceder à região crítica do algoritmo. Caso estivéssemos a aceder ao mesmo tempo a valores de clusters tinhamos o risco de criarmos situações de corrida de threads, algo importante a evitar. Deste modo o grupo testou cláusulas como ***pragma omp atomic pragma omp critical*** para cada instrução dentro do ciclo porém estas soluções demonstraram ser bastante pesadas e foram descartadas. Outro construtor abordado para a otimização da paralelilzação foi o ***reduction*** com o operador '+' para os valores de somatorio de cada cluster algo que também não resultou.

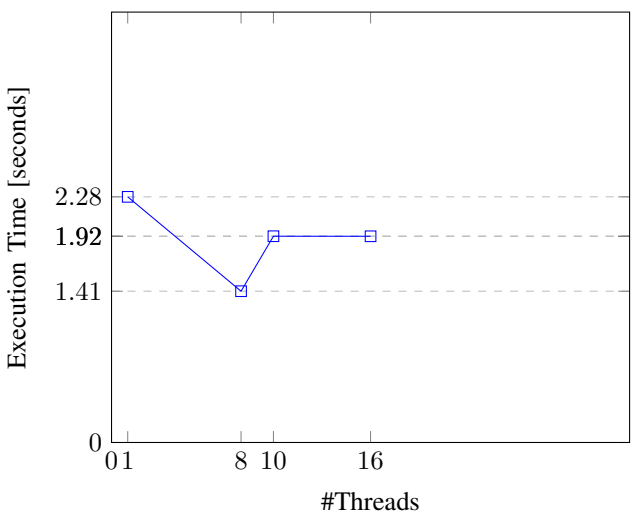
V. Alternativa mais fiável

Na escolha da alternativa mais fiável, o grupo decidiu optar apenas pela cláusula ***pragma omp parallel for schedule(static) num_threads(t)*** no segundo bloco analisado. Por outro lado foi decidido não paralelizar o terceiro bloco, uma vez que segundo os resultados obtidos no tempo de execução do algoritmo, estes não foram satisfatórios. As tentativas de paralelização do terceiro esbarraram nos resultados uma vez estes possuíam valores diferentes dos representados no enunciado fornecido pelos docentes.

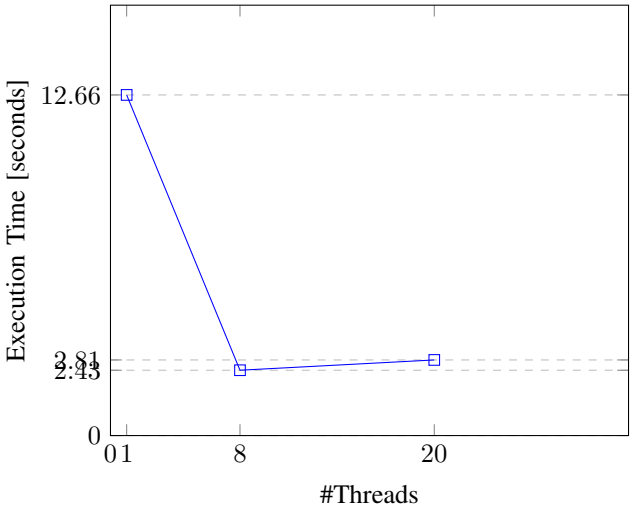
VI. Análise de Escalabilidade e Análise de Desempenho de Implementação

No conjunto, foi possível verificar que a versão paralela do algoritmo k-means tem um melhor desempenho global. Além disso, é também uma solução escalável até certo número de threads. Foram criados dois gráficos com a representação dos testes feitos e resultados obtidos que nos permite compreender a escalabilidade do algoritmo paralelizado implementado. É possível verificar que no caso de 4 clusters a partir de 10 threads a velocidade do algoritmo começa a ficar constante, algo que também acontece por volta das 8 threads no caso de 32 clusters. É de notar também a importância que a alteração de estrutura de dados para armazenamento de valores dos pontos e clusters possui, uma vez que as estruturas novas (arrays) permitem guardar valores interpretados como um bloco contíguo de memória, melhorando os acessos e a localidade de cache. Embora o grupo esteja otimista consoante os tempos obtidos é possível concluir que a não paralelização do terceiro bloco analisado impede talvez uma otimização tanto na velocidade como na escalabilidade do algoritmo.

Análise de Escalabilidade para 4 Clusters



Análise de Escalabilidade para 32 Clusters



VII. Conclusão

Neste segundo trabalho da unidade curricular de Computação Paralela foram estudados parâmetros acerca de paralelização, desde o estudo de variáveis partilhadas até concurrência. A implementação do algoritmo anteriormente criado possuía bastantes pontos a melhorar, principalmente em estruturas de dados que facilitavam o acesso a memória. Porém o grupo pensa que os resultados foram satisfatórios, toda a análise feita em diferentes métricas promoveu a aprendizagem de conteúdo relacionado com a paralelização. Percebemos então que, a inserção de pragma deve ser criteriosa de modo a diminuir o tempo de execução do código e que por vezes, não é vantajosa e pode até sair muito cara computacionalmente, como é o caso do pragma omp critical. Concluimos também que o suporte dos pragma com structs é muito limitado, como é o caso das reduções. A utilização da interface *openMP* permitiu pôr em prática o conhecimento de todo o conteúdo abordado nas aulas práticas e teóricas de Computação Paralela. Em conclusão, o grupo percebe que existem aspetos a melhorar tanto a nível de implementação como a nível teórico, algo que deverá ser melhorado para o próximo projeto da unidade curricular.

References

[1] https://datasciencelab.wordpress.com/tag/lloyds-algorithm/