Relatório do Trabalho Prático 2

Computação Paralela, Unidade Curricular integrada no Mestrado Em Engenharia Informática, Universidade do Minho

Braga, 18 de novembro de 2022

Bruno F. M. Pereira (PG 50271), João
P. F. Delgado (PG 50487)

Abstract

Este relatório, referente ao trabalho prático número dois da Unidade Curricular de Computação Paralela, teve como principal objetivo a exploração da paralelização do algoritmo *K_Means*, desenvolvido na fase anterior, cujo principal objetivo incide na redução do tempo de execução do programa. Será, assim, apresentada uma explicação sobre os diversos passos que levaram à realização desta tarefa, bem como os resultados obtidos.

I. Introdução

A computação paralela é um dos paradigmas mais relevantes de programação, face aos grandes desafios e restrições de nível físico em aumentar a capacidade de computação dos processadores. Este tipo de paradigma permite dividir problemas de grande complexidade em problemas mais pequenos que podem ser executados em paralelo, aumentando a eficiência dos programas e do uso dos recursos disponíveis.

De tal modo, uma implementação do algoritmo kmeans apresenta uma grande carga computacional que poderá ser dividida em partes mais pequenas, que com o devido controlo, poderão executar em paralelo, permitindo uma maior eficiência e rapidez nos cálculos durante cada iteração deste algoritmo.

Portanto, para realizar a aplicação do paradigma de computação paralela à implementação do kmeans desenvolvida durante o trabalho TP1 entregue, primeiro deve proceder-se à analise da carga computacional das diferentes partes do código que implementa o algoritmo. Esta análise servirá para identificar quais os blocos que apresentam maior carga e que, por isso, poderão beneficiar da paralelização de código, contribuindo para um ganho de performance em todo o programa.

Finalmente, será realizada uma análise de ganhos, apresentando os resultados obtidos na aplicação da paralelização comparando com aquilo que seriam os ganhos ideais numa situação perfeita em que todo o código é paralelizável.

II. Profilling do programa

De forma a conseguir perceber onde é que se localizava a maior parte da carga computacional do programa, foi feito o profilling do mesmo, com a ajuda da ferramenta *perf*. Com a *flat view* fornecida pelo *perf report*, pudemos concluir que 83% da carga do programa estava na função *assign_point_to_cluster*, onde em cada uma das iterações do *k_means* era feita a iteração dos pontos todos, ou seja, este ciclo acabava por ser executado 10 milhões de vezes em cada iteração *k_means*. Assim, o grupo achou adequado aplicar paralelismo a esta secção do código. Assim, paralelizando esta secção de código, estaríamos a paralelizar 83% do código aproximadamente.

Pela lei de *Amdahl*, o ganho possível após esta otimização, poderá ser, no máximo: G_max = 1/f, sendo f a percentagem de código sequencial. Sendo f = 0.17, neste caso, G_max = 5.88, em teoria. Ou seja, se o tempo de execução da secção paralela aumentasse, por exemplo, 10 vezes, o tempo de execução do programa no geral só iria aumentar 3.95 vezes. Assim, a diminuição do tempo de execução do programa está limitada pela secção de código não paralelizável, como é o caso da função *k_means*. Esta não pode ser paralelizada, uma vez que o estado inicial de uma iteração depende do estado final da iteração anterior.

III. O que foi paralelizado

Após a análise do *profilling* do *perf*, consegue concluir-se que a maior carga computacional da implementação do *kmeans* encontra-se na função *assign_point_to_cluster*, função que processa um ponto e o atribui a um *cluster*, atualizando as somas das coordenadas dos pontos de cada *cluster* e o número de pontos em cada *cluster*.

Posto isto, a atenção focou-se sobre a função *iterate_points* que itera todos os pontos, chamando para cada um deles a função *assign_point_to_cluster*, que irá realizar, para cada ponto, o processamento explicado anteriormente. Esta função apresenta um ciclo de N iterações (sendo N o número de pontos a processar), pelo que foi tomada a decisão de paralelizar as iterações deste ciclo utilizando para isso o contrutor *parallel for*.

A decisão de paralelizar este ciclo apresentava o risco da ocorrência de *data races*, já que as *threads* estariam a alterar dados de memória partilhada, pelo que foi criada uma nova estrutura auxiliar. Esta estrutura teria as somas da coordenadas x e y de cada cluster assim como o total de pontos, sendo que cada *thread* terá o seu próprio conjunto de estruturas, de forma a não gerar *data races*. De seguida, estas estruturas foram agrupadas numa matriz, sendo que cada linha tinha K estruturas (uma por cluster), e existiriam tantas linhas como *threads* a executar.

Esta estrutura permitia que cada *thread* acedesse apenas a uma linha da matriz, sendo essa linha indexada pelo id da *thread*. No final do processamento de todos os pontos, são atualizados os dados dos clusters somando os valores acumulados por cada *thread*.

O escalonamento usado foi o definido por defeito do construtor *#omp parallel for*, que é o escalonamento estático, onde o trabalho das iterações do ciclo é dividido por todas as *threads* que o vão executar de forma homogénea. Este foi o escalonamento escolhido, uma vez que cada iteração do terá

aproximadamente a mesma carga computacional, seja qual for o ponto analisado.

IV. Análise de Resultados

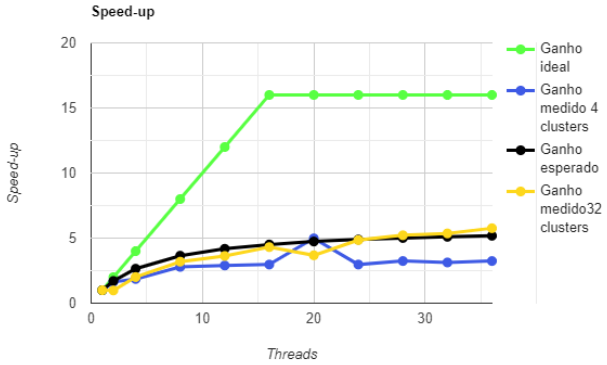
A fase da análise de resultados teve como objetivo a comparação entre os resultados obtidos na análise teórica (fase de *profilling*) com os resultados obtidos nos testes feitos no *search*, para 4 e 32 clusters.

Nas tabelas que se seguem, apresentam-se os resultados obtidos para os testes solicitados.

Nº Threads	Tempo de execução (em segundos)
1	2.79
2	1.74
4	1.55
8	1.00
12	0.96
16	0.63
20	0.56
24	0.94
28	0.86
32	0.89
36	0.86

Nº Threads	Tempo de execução (em segundos)
1	15.90
2	16.23
4	7.91
8	4.98
12	4.37
16	3.68
20	4.32
24	3.27
28	3.05
32	2.9
36	2.76

Para comparação com a análise teórica, foi também desenhado um gráfico que relaciona o número de *threads* com a medida de *Speed-Up*, a medida usada para avaliar a execução do programa paralelo.



Neste estão representadas 3 linhas. A verde, a linha do ganho ideal, que seria possível atingir caso 100% do código fosse paralelizável e não houvesse qualquer limitação na paralelização do mesmo. A preto, está representado o ganho que seria esperado obter à medida que o número de *threads* vai aumentando, valores calculados a partir da lei de *Amdahl*. Por fim, a azul estão representados os valores que foram obtidos nos testes do cluster, que constam nas tabela com os resultados obtidos para 4 *clusters*.

Quanto ao #I, na versão sequencial do código o seu valor era 17 055 415 150, enquanto na versão paralela foi 18 116 914 022, como era de esperar, uma vez que a versão paralela, para além das instruções já existentes na versão sequencial, existem também as instruções de criação e gestão das *threads*.

V. Conclusões

Primeiramente, após verificar as medições dos tempos de execução da implementação do algoritmo *k_means*, antes e depois de paralelismo é notório o ganho de performance, já que os blocos de código que representavam a maior parte da carga computacional do programa foram paralelizados.

Face ao que foi apresentado na análise de resultados, podemos observar que os resultados obtidos nem sempre atingiram aquilo que eram os resultados esperados, os quais foram calculados através da lei da *Amdahl*. Uma possível explicação para esta discrepância poderá estar relacionada com limitações na largura de banda da memória, que não permita obter todos os dados à taxa que estes são pedidos, fazendo com que as diferentes *threads* tenham de aguardar pela resposta da mesma.

Para além disso, durante a realização das medições com o *perf* no *cluster* foi notória uma grande variação nos resultados obtidos, essas variações provavelmente estão relacionadas com o número de utilizadores a tentar executar *jobs* no *cluster*, pelo que resultados mais fidedignos poderiam ser obtidos num cenário perfeito em que mais nenhum utilizador estivesse a tentar executar *jobs* no *cluster*.

Assim, depois de feitos os testes, estes mostraram que o número ideal de *threads* para a execução do algoritmo é 20 e 36 para 4 e 32 clusters, respetivamente, já que é nestes pontos que foi registado o maior ganho.

Finalmente, é relevante apontar que o "ganho ideal" apresentado na análise de reultados nunca poderá ser atingido, já que não é possível paralelizar 100% do código.