Trabalho Prático 2

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 2 da unidade curricular de Computação Paralela, que tem como objetivo a exploração de paralelismo no programa criado no primeiro trabalho, de modo a diminuir o seu tempo de execução.**

***Index Terms*—computação paralela, OpenMP, paralelismo, linguagem C, algoritmo k-means**

I. Introdução

Neste projeto foi desenvolvido em C uma nova versão do programa desenvolvido no trabalho prático anterior, que faz uso de paralelismo. O programa implementa 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, sendo que nesta fase se foca mais no uso da biblioteca OpenMP para tornar o programa paralelo.

II. Alterações no código

O código desenvolvido no primeiro trabalho prático sofreu algumas alterações de modo a cumprir os novos requisitos e objetivos propostos. Nomeadamente a possibilidade de alterar os valores de pontos, clusters e threads, através dos inputs dados na linha de comandos, outro objetivo proposto foi melhorar o tempo de execução de modo a torná-lo o mais rápido possível, por fim foi também necessário adaptar a Makefile para que o utilizador tenha a escolha de correr o programa de modo sequencial ou paralelamente.

Para tal, começamos por alterar as variáveis globais, variáveis *hard coded*, que definiam o número de pontos e clusters, de modo que seja possível o utilizador definir as mesmas, deixando estas de serem *hard coded*. Tivemos também o cuidado inicial de permitir que o programa possa correr de modo sequencial ou paralelamente, alterando a Makefile e tendo em atenção no código desenvolvido, nomeadamente o uso de várias threads ou apenas uma.

Com o propósito de tornar o programa com um tempo de execução menor destruímos as estrutura de dados do ponto e do cluster, passando tudo para arrays, para cada ponto temos: um array para o cluster, um array para a coordenada x e outro array para a coordenada y. Em relação ao cluster passamos a ter: um array para o tamanho, um array dos somatórios da coordenada X dos pontos , um array dos somatórios da coordenada Y dos pontos, um array para a coordenada X do centróide e outro array para a coordenada Y do centróide. Consequentemente com estas alterações tivemos que modificar um pouco do resto do código existente.

O bloco de código com maior carga computacional é o ciclo onde são percorridos todas as amostras e é feito o cálculo da distância entre os pontos e os centróides dos vários clusters. Na primeira versão do programa eram alteradas as variáveis dos somatórios das coordenadas dos pontos e o tamanho do cluster dentro deste ciclo, porém ao desenvolver a versão paralela do programa esta alteração das variáveis iria implicar um controlo de concorrência, o que resultaria num tempo de execução maior do que se a alteração fosse feita noutro ciclo independente deste. Para este ciclo onde é feito o cálculo da distância entre os pontos e os centróides, são criados os vários fios de execução que permitem dividir o trabalho pelas várias threads. No final do ciclo, depois de todas as threads acabarem o seu trabalho e forem calculados os novos clusters para os pontos, o programa irá voltar a ter apenas um fio de execução e é feito um novo ciclo sobre todos os pontos, onde são atualizadas as variáveis dos clusters (somatório das coordenadas dos pontos do cluster e tamanho do cluster).

Por fim, nesta fase o critério de paragem foi também alterado, sendo apenas necessário que o programa chegue às 20 iterações tendo resultados iguais ou semelhantes aos disponibilizados. Tendo em conta a este novo critério, foi removido o teste que verificava se o algoritmo tinha convergido, ou seja, nenhum ponto tinha mudado de cluster.

III. Análise de escalabilidade

Tal como requisitado no enunciado o programa tem de ser escalável, ou seja, permitir um número de valores diferentes de clusters, amostras e fios de execução. Esta escalabilidade é suportada nos dois modos do programa supracitados, sequencial e paralelo. É possível concluir que o programa paralelo é mais escalável que o sequencial, pois pode dividir o trabalho que terá que executar pelos vários fios de execução, enquanto que

o programa sequencial terá que executar todas as instruções do algoritmo de maneira ordenada.

Contudo, a escolha do número de threads a utilizar no programa paralelo tem de ser cuidadosa, pois se forem utilizadas poucas threads o programa não irá tirar partido máximo do paralelismo e não irá executar no melhor tempo de execução possível. Pelo contrário, se forem utilizadas threads a mais, estas podem tornar o programa mais lento, por causa do *overhead* gerado nas trocas de contexto entre threads.

IV. Medição do perfil de execução

Para efeitos de teste, o programa foi executado dos dois métodos: sequencial e paralelo. No modo paralelo foram testadas duas, quatro, oito, doze, dezasseis e vinte threads. Quanto ao número de clusters e amostras, foram feitos testes para 4 e 32 clusters com 10 mil milhões de pontos. Os valores das colunas do número de ciclos de relógio, número de instruções e tempo de execução resultam de uma média de cinco execuções do programa.

	CC (mil milhões)	#I (mil milhões)	CPI	Texec (s)
Sequencial 4 clusters	7.2	18.3	0.4	2.43
Sequencial 32 clusters	40.9	94.8	0.4	13.39
Paralelo 4C e 2T	7.3	17.7	0.4	1.64
Paralelo 4C e 4T	7.6	17.9	0.4	1.25
Paralelo 4C e 8T	8.0	18.0	0.4	1.06
Paralelo 4C e 12T	8.4	18.2	0.5	1.00
Paralelo 4C e 16T	8.9	18.4	0.5	0.96
Paralelo 4C e 20T	11.7	19.4	0.6	1.03
Paralelo 32C e 2T	42.4	86.9	0.5	7.47
Paralelo 32C e 4T	42.7	87.1	0.5	6.22
Paralelo 32C e 8T	43.4	87.3	0.5	2.52
Paralelo 32C e 12T	43.5	87.4	0.5	1.97
Paralelo 32C e 16T	44.0	87.6	0.5	1.68
Paralelo 32C e 20T	46.5	88.5	0.6	1.78

Analisando a tabela, é possível verificar que para o programa paralelo o número ideal de threads é dezasseis, pois obtém melhores tempos de execução que as versões com menos threads. O aumento da quantidade de threads para além deste número irá causar tempos de execução semelhantes, porém o número de instruções e de ciclos de relógio aumenta. Este aumento deve-se ao facto de o computador necessitar trocar de contexto entre threads, o que causa o *overhead* referido anteriormente.

V. Balanceamento de carga

Relativamente ao balanceamento da carga pelas várias threads este é feito de forma automática quando é executado o comando *#pragma omp parallel for*. Por defeito, o que esta instrução faz é a separação do trabalho pelas várias threads de uma forma equilibrada, neste caso, irá dividir o ciclo em várias partes de acordo com o número de fios de execução. Por exemplo, se o programa receber como input que terá de criar quatro threads este irá dividir o ciclo em quatro partes iguais e atribuir a cada thread uma parte para executar.

Para alterar este comportamento é utilizado o *scheduling*. Como as iterações do ciclo do programa executam em tempos muito semelhantes, o *schedule* estático é melhor pois tem pouco *overhead*. Ao utilizar o *schedule* dinâmico este irá atribuir a cada thread a próxima iteração não executada, isto irá impedir a vetorização e causar *overhead* desnecessário.

Também podem ser utilizados *chunks*. Cada thread executa um número definido de iterações, um *chunk*, e quando acabar passa para o próximo *chunk*. Aqui é necessário ter um cuidado especial pois se a divisão for muito pequena, o programa poderá não tirar o melhor partido da vetorização, tal como no *schedule* dinâmico. Aumentando cada vez mais o tamanho do *chunk* o *schedule* torna-se estático.

Por fim também existe o *schedule* guiado. Vão ser utilizados *chunks*,sendo que estes começam com um tamanho grande e ao longo da execução do programa o tamanho vai diminuindo. Este tipo de *scheduling* funciona melhor quando o trabalho entre as threads não é balanceado, o que não é o caso deste programa.

VI. Conclusão

Este segundo trabalho prático permitiu-nos consolidar de um modo mais conciso os temas abordados no trabalho anterior e os novos temas abordados para este trabalho, pois foi necessário, além das aulas teóricas e práticas, um estudo mais pormenorizado sobre flags e todas as funcionalidades do OpenMP, de modo a obter o melhor resultado possível. Concluímos também que todo o trabalho permitiu-nos adquirir um maior conhecimento e experiência sobre o melhor manuseamento e construção do código de modo a tornar o programa mais eficiente, conhecimento este que se tornará útil em trabalhos futuros.

References

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

[2] OpenMP ARB, "OpenMP 4.0 API C/C++ Syntax Quick Reference Card," https://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf