Faculdade de Ciências da
Universidade do Porto

Computação Paralela

All-Pairs Shortest Path Problem

José Pedro Sousa
201503443

8 de Novembro de 2019

1 Intrudução

1.1 Ambito

Este trabalho consistiu numa implementação de um problema dada na unidade curricular Computação Paralela e que permite ao estudante adquirir e praticar a experiência na programação em computação paralela para ambientes de memoria partilhada usando uma libraria da linguagem de programação C chamada MPI.

1.2 Descrição do Problema

O objetivo deste projeto é uma implementação ao problema *All-Pairs Shortest Path Problem* que basicamente dando um grafo dirijido G=(V,E) em que V é um conjunto de vertices(nos) e E é um conjunto de arestas(ramos) entre os nós, em que a finalidade é determinar para cada par de nós (vi, vj) o caminho minimo que começa no vetice vi e termine no vertice vj. Um caminho é uma sequência de arestas(ramos) em que o vertice final e o vertice inicial de dois consecutivos ramos são os mesmos. Um ramo pode ser visto como um tuplo da forma (vi, vj, c) em que vi é o vertice inicial(origem), vj é o vertice final(destino) e o c é o tamanho do ramo entre os vertices vi e vj. O tamanho de um caminho será a soma de os ramos entre vi e vj.

2 Algoritmo

2.1 Ideia

Para que a implementação da solução seja possivel, o dado grafo dirigido irá ser representado numa matriz D1 com dimensão NxN em que cada emlemento (i,j) da matriz representa o tamanho do ramo entre vertice vi e o vertice vj. O valor zero representa que a coneccção entre os dois vertices caso não existe.

Para solucionar este problema através da implementação de matrizes, irá ser usada um calculo modificado de multiplicação entre matrizes. Esta multiplicação é uma adaptação da forma como se multiplica as matrizes, na qual as operações de multiplicação e soma sao substituidas por operaçoes de soma e minimo respetivamente.

A ideia do algoritmo é modificar o modo de executar este tipo de multiplicações de matrizes que foi mencionado, a uma certa quantidade de vezes para calcular os caminhos minimos entre os pares de vertices. Para que isto seja possivel são usados dois algoritmos: *Repeated Squaring Algorithm* e *Fox Algortihm*.

2.1.1 Repeated Squaring Algorithm

Este algoritmo utiliza uma propriedade simples que melhora o numero de vezes que é necessario fazer a multiplicação de matrizes para alcançar a solução desejada. Por outras palavras, este algoritmo permite obter a matriz Df da matriz D1(matriz inicial) construindo sucessivamente matrizes D2=D1xD1, D4=D2xD2, D8=D4xD4,... até Df=DgxDg com f $\geq Neg < N$.

2.1.2 Fox's Algorithm

Fox é um algoritmo usado para fazer parte na multiplicação de matrizes sendo eficiênte num ambiente de computação paralelo. No contexto deste problema, o algoritmo *fox* foi alterado para realmente calcular os caminhos minimos em vez de multiplicar numeros. Este algoritmo ira dividir a matriz com tamanho N em submatrizes para que cada processador P possa calcular a submatriz soluçao de tamanho N/Q.

No entanto para que este algoritmo seja possivel aplicar irá ter que obdeccer algumas regras relativamente ao numero de processaores e ao tamanho da matriz inserida. Dado um tamanho N da matrix inserida e P numero de processadores, existe Q tal que P=Q*Q e N mod Q seja igual a zero.

Para computar o resultado de cada sub-matriz, cada processador irá ter que tracar informação com o outro processador na mesma linha e na mesma coluna da sub-matriz. Aplicando assim sucessivamente estes dois algoritmos (Fox's e Repeated Squaring) iremos ter a matriz soluçao Df ao problema.

A solução do problema sera representado numa matriz final Df com dimensoes NxN em que cada elemento (i,j) da matriz corresponde ao tamanho do caminho minimo de todos os possiveis caminhos entre o vertice vi e o vertice vj.

2.2 Implementação

A implementaçao que foi usada na parte da contruçao do algoritmo *Fox* foi baseada pela implementação do livro *Parallel Programming With MPI* do autor *Peter S. Pacheco*. Na implementação foi criada dois ficheiros em que num é onde se insere o *main* do programa e noutro para as *structs*. Foi criada uma *struct* para a grid($GRID\_TYPE$) que irá facilitar o acesso a informações do processador dentro da topologia da grid.

O algoritmo *Repeated Squaring* foi implementado globalmente no código, ou seja, qualquer processo tem o mesmo acesso ao codigo para calcular este algoritmo. Em cada iteraçao do calculo da multiplicaçao de cada matriz é chamado o algoritmo Fox. Antes de cada iteração devemos garantir que

cada processador tem a sua submatriz. Depois do calculo do algoritmo Fox, o resultado da submatriz com este algoritmo sera a submatriz para a iteração seguinte.

No final do calculo o processador *ROOT*(processador com rank 0) irá agregar todas as submatrizes de todos os processadores que calcularam e irá juntar numa só matriz que será a matriz final, ou seja, a matriz resultado(sera inprimido para o output, usando a função *print_matrix()*).

Para além das funções usadas para o calculo do algoritmo Fox, tambem foram criadas outras funções para a alocação de memoria, copias de matrizes, criação da matriz e da grid, etc. As mais relevantes destas funções foram a *create_grid()* que faz a configuração e inicialização da grid inicial utilizando a struct *GRID_TYPE*, também a função *flag_func()* que irá validar se o algoritmo pode ser implementado de acordo com as regras mencionadas na secção 2.1.2, caso não seja possivel, uma menssagem de erro irá aparecer no output e terminará o programa. Outra função que foi construida de maior relevancia foi o *operation_multiply()*. Esta função implementa o calculo especial para a multiplicação de matrizes mencionada na seccçao 2.1.

3 Performace

Reativamente há execução do programa, foi testado usando um *cluster* de computadores com cada um 4 cores, sendo num total maximo operacional de 64 cores.

Para compilar este programa basta executar o comando *$make* de seguidamente para executar ou correr o programa *$mpirun –hostfile hostfile -np P fox*, em que o P é o numero de processadores que pretende executar.

No final da execução do programa, irá ser imprimido para o output do terminal uma contagem do tempo de execução e a matriz final da execução. Caso não seja possivel a execução do algoritmo uma menssagem de erro ira ser imprimida.

Infelizmente o programa tem algumas limitaçoes, uma delas e mais relevante foi que durante a execuçao de alguns testes para matrizes de grandes dimensões e com numero de processadores relativamente pequeno, a execuçao nao parava e entraria num ciclo infinito de execuçao.

3.1 Resultados

O programa foi testado para varias matrizes de varias dimensões(6,300,600,900) com varios numeros de processadores(1, 4, 9, 16, 25, 36, 64). Foram feitos dois testes com mas mesmas configurações, mas em horarios diferentes de utilização das maquinas. Estes foram os seguintes resultados(em ms):

Teste nº 1							
Dim:CPU	1	4	9	16	25	36	64
6	0.123	0.075	1.246	ANA	ANA	4.371	ANA
300	NF	NF	615.9	435.809	448.663	407.106	ANA
600	NF	NF	NF	NF	3067.375	2901.086	1668.257
900	NF	NF	NF	NF	NF	NF	ANA

Teste nº 2							
Dim:CPU	1	4	9	16	25	36	64
6	0.134	0.093	1.545	ANA	ANA	4.454	ANA
300	NF	NF	622.351	433.63	453.236	410.208	ANA
600	NF	NF	NF	NF	3097.365	2060.921	1643.242
900	NF	NF	NF	NF	NF	NF	ANA

NF: Nao Finalizado
ANA: Algoritmo Nao Aplicado
Dim: Dimensão da matriz

4 Conlusão

Concluimos assim de acordo com os resultados que para matrizes de maiores dimensões, ao fazer o acrescimo de numero de processadores executados, iremos ter uma diminuição de tempo de execução, ou seja, uma maior rapidez e eficiencia na execuçao do algoritmo para calcular a solução ao problema *All-Pairs Shortest Path Problem.*