All-Pairs Shortest Path Problem

Artur Manuel Pascoal Ferreira
up201302914

9 de Novembro de 2019

Conteúdo

Capítulo 1

Implementação do Algoritmo

A ideia inicial para a implementação do algoritmo foi usar o código já existente (do livro dado no enunciado) e construir o resto do programa à volta disso. Dito isto, foram feitas algumas alterações ao código inicial. Por exemplo, a função `Set_to_zero` foi mudada para `Set_to_max` isto porque para obter a distância mínima, inicialmente o *array* deverá ter valores altos e não baixos. Os parâmetros da função `Fox` foram mudados para receber *arrays* de inteiros em vez de um objecto de uma estrutura de dados escrita no programa.
Na leitura da matriz é usada a função `Transform` para transformar o valor 0 em `MAX` (definido no início), caso a posição desse valor não fosse na diagonal principal. Para imprimir a matriz final é usada a função `Transform_inverse` que, tal como o nome indica, faz o inverso de `Transform`.
Como o objecto usado para guardar os valores é um *array* de inteiros, de a forma obter, mudar e actualizar um valor numa matriz foram criadas funções auxiliares que calculam a posição onde se encontra o valor e retornam/mudam o mesmo.
Para os diferentes processos poderem trocar valores entre si, temos a estrutura `GRID_INFO_TYPE` e a função `Setup_grid` (ambas já no código do livro). Os elementos relevantes da estrutura consistem em 3 comunicadores (um para todos os processos, um para cada linha e outro para cada coluna), o *rank* do processo e as coordenadas do processo. Enquanto que, a função serve para configurar a estrutura em cada processo. Primeiro é criado o comunicador com uma topologia cartesiana para os processos usando o `MPI_Cart_create` e partir deste são gerados os comunicadores para cada linha e coluna usando o `MPI_Cart_sub`.
Com a configuração dos comunicadores feita, começa o algoritmo principal (`Min_plus_matrix_mul`) que contém um *loop* onde é executada a função `Fox`, multiplicando uma matriz com ela mesma e usando a matriz resultante para fazer o mesmo na próxima iteração. Na função `Scatter_matrix`(executada imediatamente antes da `Fox`), o processo com *rank* 0 fica encarregue de enviar as sub-matrizes para o processo correspondente. Na

função `Gather_matrix`(executada imediatamente depois da `Fox`), cada processo envia a sua sub-matriz final (`local_C`) para o processo com *rank* 0, sendo que este constrói uma nova matriz com os resultados recebidos.
A função `Fox`, começa por calcular os *ranks* dos processos para quem vai receber e enviar a sua segunda matriz (`local_B`) da "multiplicação" em cada iteração do *loop* do algoritmo. Em cada uma das iterações é calculado o processo a fazer o *broadcast* da sua matriz principal (`local_A`) aos processos da mesma linha. Após isto, o *"broadcaster"* executa o `MPI_Bcast` e calcula a "multiplicação" das matrizes enquanto que os restantes recebem a matriz do *"broadcaster"* e calculam também a sua "multiplicação". No final de cada iteração, cada processo executa o `MPI_Sendrecv_replace` isto é , envia a sua segunda matriz e substituí a mesma pela a que recebe.

Capítulo 2

Avaliação dos Tempos de Execução

Para cada matriz exemplo dada, o programa foi executado 3 vezes e os resultados apresentados são a média dos mesmos, de forma a remover possíveis anomalias (e.g. outros utilizadores a utilizar as máquinas, problemas de rede, etc) .

	6	600	900	1200
x Sequencial	0	78249	272259	795248
y P = 1	0	79435	277649	808685
x - y	0	1186	5390	13437

Tabela 2.1: Tempos médios de execução (Sequencial *vs* P = 1)

Na tabela 2.1 podemos ver existe diferença entre correr o programa sequencialmente e em paralelo com P = 1 (pelo menos para as matrizes dados no enunciado) visto que a diferença dos dois tempos aumenta e de modo não linear.

	6	600	900	1200
1	0	79435	277649	808685
4	0	21045	72684	195676
9	1	17635	51022	123676
16		16845	43400	98378
25		16311	41026	89100
36	8	16275	39417	82481

Tabela 2.2: Tempos médios de execução em milissegundos

A tabela 2.2 mostra que para N = 6 o tempo aumenta com o número de processos isto acontece porque o programa gasta mais tempo a sincronizar os processos do que a calcular as "multiplicações"das matrizes. Para todas as outras dimensões o tempo reduz, o que é esperado.

	6	600	900	1200
1	1	1	1	1
4	0.53	3.77	3.82	4.13
9	0.08	4.50	5.44	6.56
16		4,72	6.40	8.22
25		4,87	6.77	9.08
36	0.02	4.88	7.04	9.80

Tabela 2.3: *Speedups*

	6	600	900	1200
4	-47	277	281	313
9	-85	19	42	59
16		5	17	25
25		3	6	10
36	-76	0	4	8

Tabela 2.4: Percentagem de *speedup* ganho(%)

Na tabela 2.3 vemos que os *speedups* aumentam com o número de processos, à excepção do N = 6 que sofre muito *overhead*. A diferença entre cada linha diminui com o aumento de processos e de modo a facilitar a visualização deste acontecimento, temos a tabela 2.4 em que cada linha representa o ganho de *speedup* (em percentagem) em relação à linha anterior,sendo que a primeira linha (P = 4) compara-se com P = 1. Aqui, podemos ver que para P = 4 é onde obtemos, sem dúvida alguma, os melhores ganhos. Este efeito seria ainda mais saliente caso tivéssemos em conta o número de máquinas necessárias para cada P e o custo delas. É de notar que na tabela 2.3, em P = 4 e N = 1200 é onde obtemos o melhor *speedup* em que este é maior que o seu número de processos. Isto é provavelmente causado pelas anomalias já referidas e não pela implementação em si.

Capítulo 3

Dificuldades Encontradas

Houve três obstáculos que consumiram uma quantidade de tempo considerável para encontrar e resolver. O primeiro foi que, ao tentar fazer a validação de P, N e Q, as dimensões do comunicador cartesiano foram mudadas para P (em vez de Q) o que levou a que fosse lançado um erro sempre que $P > 1$. O segundo foi que, como as matrizes são iguais, pensava que seria apenas necessário alocar espaço para uma matriz local de cada processo e enviar como parâmetros da função Fox, a matrix local_A duas vezes. Isto apenas funciona para $P = 1$ pois o processo quando troca uma matriz, troca pela mesma. Diria que isto resultou de fazer *copy-paste* da função `Fox` a partir do livro dado no enunciado do trabalho o que leva a não perceber exactamente o que está escrito. O terceiro erro foi assumir que os métodos `Scatter` e `Gather` do MPI, ao usar um comunicador cartesiano, distribuiriam os dados correctamente pelos processos mas infelizmente, esse não é o caso.
Para finalizar, gostava de sugerir que houvessem mais métricas (dadas nas aulas ou apontadas no enunciado) de forma a compreender melhor os benefícios e prejuízo ao paralelizar uma tarefa. Um possível exemplo seria avaliar a relação custo/*performance*.