Instituto Superior Técnico

MEEC

Computação Paralela e Distribuída

Projeto, Serial & OpenMP

Estudantes:	*Número:*
João Almeida	78799
Daniel Guerra	78885
Lino Pereira	87751

7 de Abril de 2017

1 Versão Serial

O primeiro desafio para o trabalho foi decidir quais as estruturas de dados a utilizar para armazenar as células vivas e mortas.

Inicialmente todas as células vivas são guardadas numa *hashtable* indexada por $h(x, y, z) = x$. Esta função de indexamento foi escolhida para facilitar a procura na parte a seguir descrita.

Para calcular que células ficam vivas e que células ficam mortas, guardar todas as posições do cubo torna-se impossível para *sizes* muito grandes, decidiu-se por adaptar a ideia de guardar apenas uma pequena parte do cubo em cada instante. Assim guarda-se apenas 3 *slices* (sendo que uma *slice* corresponde a todas as posições do cubo com um idêntico x): a *slice* correspondente ao valor de x a analisar e as 2 *slices* adjacentes, para podermos ter disponíveis os vizinhos das células da *slice* a ser analisada que se situam em $x+1$ e em $x-1$ (devido ao *wrap*, é necessário também guardar as duas primeiras *slices* do cubo, uma vez que a última slice vai influenciar a primeira, esta só é avaliada na última iteração). Com esta abordagem evita-se fazer procuras em listas pelos vizinhos, aumentando a performance do programa. Depois de analisadas todas as células vivas desliza-se o conjunto de *slices* uma posição para a direita.

Cada posição das *slices* corresponde a uma célula e a cada célula é atribuído um número que simboliza o seu estado. Inicialmente todas as células vivas são inicializadas com o valor -1 e todas as células mortas com 0. Ao analisar uma célula viva é incrementado o valor presente na posição de todos os vizinhos mortos e quando este se torna maior que 2 a sua posição é guardada numa lista para sinalizar que tal célula é candidata a ficar viva. Quando o valor de uma célula morta se torna maior do que 4, a sua posição é retirada da lista de candidatas a ficar vivas. Para verificar se uma célula viva permanece viva ou se morre é utilizado um contador que conta o número de células vivas na vizinhança. Se o número for entre 2 e 4 a célula permanece na *hashtable*, caso contrário esta é retirada da mesma. Quando uma *slice* sai da "janela deslizante"a lista com as células candidatas a se tornarem vivas é concatenada na lista da respetiva posição da *hashtable*. No fim de cada geração (correspondente à passagem da janela por todo o cubo) a *hashtable* contém todas as células vivas.

2 Versão OMP

Para paralelizar o nosso programa decidimos seguir a abordagem de usar mais "janelas deslizantes". Para tal, decidiu-se que cada *thread* iria controlar a sua própria janela. Assim, dividindo a dimensão x do cubo pelo número de *threads*, criam-se vários segmentos de cubo aproximadamente iguais, em cada um é tratado por um diferente *thread*.Para reduzir a criação e destruição de equipas de *threads*, estes são criados apenas no início do programa e destruídos no fim, em vez de cria-los e destruí-los em cada geração.

Como em cada segmento do cubo as células são independentes das células de outro segmento, exceto nas extremidades do mesmo, a atenção foi focada nestas mesmas extremidades, uma vez que o resto de cada segmento se comportaria como um cubo que pode ser analisado como explicado em 1. Para contornar este facto são guardadas as 2 primeiras *slices* do segmento de cada *thread*, uma vez que estas vão influenciar ou ser influenciadas

pelas última *slice* do *thread* anterior. Para tal, todos os *threads* têm de saber se o *thread* seguinte já acabou de processar as 2 primeiras *slices*, uma vez que não pode existir *data races*. Além disso, antes de avançar para a próxima geração é necessário garantir que todos os *threads* acabaram de processar completamente a geração corrente.

3 Resultados

Tabela 1: Tempos de execução (s)

size	Células vivas	Gerações	Serial	OMP 1 *thread*	OMP 2 *threads*	OMP 4 *threads*	OMP 8 *threads*
5	50	10	0,006	0,008	0,008	0,009	0,007
20	400	500	1,023	1,130	0,715	0,697	0,723
50	5k	300	25,605	25,797	14,690	8,560	5,502
150	10k	1k	0,467	0,450	0,512	0,415	0,333
200	50k	1k	1,470	1,449	1,365	1,033	0,777
500	300k	2k	25,212	24,986	13,780	7,324	6,205

Tabela 2: Speedups

size	Células vivas	Gerações	Speedup 1 *thread*	Speedup 2 *threads*	Speedup 4 *threads*	Speedup 8 *threads*
5	50	10	0,75	0,75	0,67	0,86
20	400	500	0,91	1,43	1,47	1,41
50	5k	300	0,99	1,74	2,99	4,65
150	10k	1k	1,04	0,91	1,13	1,40
200	50k	1k	1,01	1,08	1,42	1,89
500	300k	2k	1,01	1,83	3,44	4,06

4 Conclusão

Sendo o principal objetivo da programação paralela a diminuição do tempo de execução de um programa conclui-se que os resultados foram bastante satisfatórios. Como seria de esperar para um mapa de jogo com dimensões reduzidas, paralelizar o programa trás despesas computacionais que levam com que o *speed up* deste seja bastante distante do ideal. Porém, para *sizes* de moderada a elevada dimensão, a despesa da criação das regiões paralelas tornava-se insignificante para o tempo de execução total, sendo o código em paralelo o mais apto para o trabalho, reduzindo significativamente o tempo de execução do programa. Como se pode observar este método de paralelização é escalável para estes *sizes*, com a excepção de 8 *threads*, mas desconfia-se do facto de os computadores do *sigma* não dedicarem os 8 cores a este programa. Os *speed ups* diferem do ideal principalmente pelo facto de os *threads* terem *slices* de diferentes densidades. Foi feita uma outra abordagem para tentar combater o referido, pondo esses *threads* ajudarem outros nas suas tarefas, mas o *overhead* de criar essas *tasks* e a quantidade de código em *critical* pioravam a performance do programa.