TÉCNICO LISBOA

MEIC-A - CPD 2016/2017
Álvaro Simões Nº 69567
João Franco Nº 75219

Introdução

O projecto consiste em tirar partido das arquiteturas com memória distribuída através da implementação paralela do algoritmo utilizado no jogo "Game of Life", sendo neste caso a variante tridimensional, neste caso usando MPI.

De forma a representar a estrutura principal do algoritmo (cubo) foi necessário proceder à modificação da estrutura feita na primeira entrega. Com o OpenMP foi utilizada uma matriz com ponteiros para listas ligada de elementos, sendo que os índices da matriz correspondem aos valores do par (x,y) e a lista dos elementos que representam os diferentes valores de z. Nesta segunda parte fomos obrigados a mudar a lista ligada de elementos z para arrays dinâmicos, de modo a facilitar a troca de mensagens, uma vez que os ponteiros não teriam o mesmo significado em diferentes máquinas. Estes arrays de tamanho dinâmico foram implementados de forma eficiente tal que o tempo de inserção e remoção é O(1) amortizado.

Descrição

Abordagem

A nossa abordagem ao problema visa percorrer a estrutura contendo os elementos vivos de modo a visitar os seus vizinhos ao mesmo tempo que faz a sua contagem e adiciona-os como elementos "mortos". No final desta contagem é feita a remoção dos elementos mortos. A forma como abordamos o problema foi a seguinte:

- Primeiramente, ao iniciar o programa, existe um processo root que é encarregue de ler o ficheiro de input e enviar as células aos outros processos.
- Em loop, de acordo com o número de gerações indicadas nos argumentos do programa, cada processador irá realizar uma troca de fronteiras, formando um conjunto de "ghost points" necessários à computação dos vizinhos.
- No passo seguinte é feita a contagem de elementos vizinhos vivos e a consequente verificação de condição de sobrevivência. Se existirem ainda gerações em falta regressamos ao passo anterior.
- Por último o processo root recebe os resultados de todos os processos e imprime os elementos vivos.

Decomposição

Foi utilizada a metodologia de foster para estruturar o nosso programa:

Primitive Task: Fazer a contagem de vizinhos de uma célula.

Partitioning: Computar para cada célula do cubo a contagem de vizinhos

Communication: Células adjacentes (x+1, x-1, y+1, y-1, z+1, z-1)

Agglomeration+Mapping: A matriz contendo as células foi partida em blocos de linhas usando a abordagem de decomposição por blocos, após a definição destes blocos da matriz foram atribuídos a cada um dos processos.

Com esta abordagem conseguimos definir para cada processo um conjunto de linhas contíguas constituindo um bloco, o que leva à diminuição de comunicações. Cada processo apenas precisa de obter a linha acima do seu bloco e a linha abaixo do mesmo. Todas as outras células necessárias já se encontram dentro do próprio bloco. Estas células formam um limite "pontos fantasmas" usados apenas para ajudar na computação dos vizinhos das células pertencentes aos blocos. Na figura Fig.1 é possível ver um exemplo deste caso.

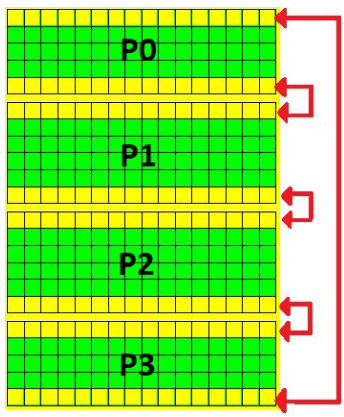

Fig.1 - Exemplo de zonas de fronteira e troca de linhas com 4 processos

Sincronização

De modo a garantir que cada processo executa de forma paralela mas correcta foram assinaladas zonas onde era necessária sincronização. Estas são: inicialmente todos os processos esperam por

receber os blocos de linhas do processo root inicializando a sua matriz; dentro do loop todos os processos precisam de esperar a chegada das linhas fronteira que necessita para a computação.

Load Balancing

No que diz respeito ao balanceamento da carga entre processos houve uma agregação de linhas em blocos da estrutura em matriz e consequente atribuição dos blocos aos processos de forma uniforme.

Resultados e Análise

Os resultados obtidos foram registados na tabela abaixo e subsequente gráfico originado a partir desta. Foram selecionados apenas 3 testes que correspondem aos que têm maior size do cubo, pois os restantes não apresentam tempos significativos de modo a poder extrair o potencial de um maior número de processos.

Tabela 1 - Tempos em segundos (s) de execução do algoritmo para diferente números de processos

	1P	2P	4P	8P	16P	32P	64P
s500e300k	48.83	24.79	14.56	8.67	3.77	1.82	1.64
s500e5M	73.76	40.98	22.53	11.46	6.02	3.65	1.69
s600e20M	467.6	231.2	123.0	62.5	37.0	18.5	17.1

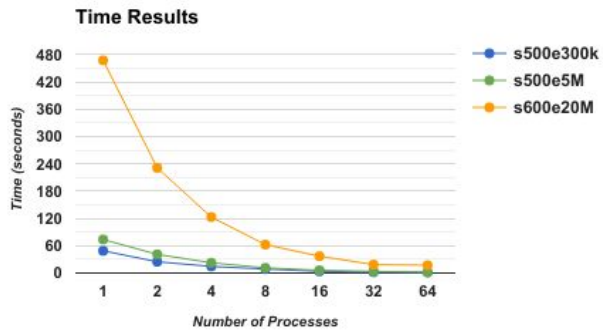

Fig.2 - Gráfico tempo(s) vs número de processadores usando os dados da Tabela 1

Tabela 2 - Speedups relativos à Tabela 1

	2P	4P	8P	16P	32P	64P
s500e300k	1.96	3.35	5.63	12.95	26.8	29.77
s500e5M	1.79	3.27	6.44	12.25	20.2	43.64
s600e20M	2.02	3.8	7.48	12.63	25.2	27.24

Antes de prosseguirmos com a análise queremos justificar que todos os testes foram executados nas máquinas do cluster da RNL e que para cada um dos inputs corremos três vezes o teste, tendo depois feito uma média de acordo com os tempos de execução.

De acordo com o gráfico e tabela dos speedups obtidos podemos concluir que no geral e até 32 processos apresentamos valores perto do ideal, no entanto para 64 processos a nossa solução não escala tão bem com o número de linhas testado. Isto deve-se ao facto do tempo de comunicação começar a dominar o tempo de computação. Uma vez que, no maior caso, temos 600 linhas para dividir pelos processos, significa que cada um irá receber aproximadamente 600/64 = 9 linhas. Isto significa que cada processo faz a computação de 9 linhas antes de partilhar as suas fronteiras.. O que verificamos neste caso é que ter 32 processadores a computar 18 linhas ou 64 processadores a computar 9 linhas não nos trás um grande melhoramento de performance derivado do overhead de comunicação. Isto significa que quando cada processo começa a ficar com um número reduzido de linhas para computar em cada geração os speedups observados começam a reduzir do ideal.

Possíveis melhoramentos

No projecto enviamos a lista de cada z correspondente a um (x, y) em MPI_sends separados. Experimentámos juntar todas estas listas num só array de modo a fazer um único send (diminuindo o tempo de comunicação), no entanto experimentalmente verificámos que o tempo que demora fazer a construção do array conjunto de um lado e a fragmentação do outro, não melhorava a overall performance do programa como nós esperávamos.

Existiam alguns outros possíveis melhoramentos que poderiam ter sido testados de modo a tentar melhorar a performance e escalabilidade. Entre eles:.

1. Em vez de distribuir de forma uniforme as linhas, era possível tentar determinar que processos tinham mais células (o que equivale a mais tempo de computação) e distribuí-las por processos que tenham menos de forma dinâmica.
2. Realizar a partição em checkerboard, em vez de de partição em linhas. Este é um método de partição mais escalável.
3. Em vez de a cada iteração trocar as fronteiras, trocar n linhas de cada vez e apenas fazer a trocas de n em n gerações. Isto diminuiria o número de mensagens enviadas em troca de alguma computação redundante em cada processador.