Relatório Do Trabalho Prático

Simulação de ecossistema

Computação Paralela 2019/2020

José Pedro Ribeiro Ferreira Pinto – 201603713

1-Descrição do problema

O problema em questão é o de simular um ecossistema simples constituído por dois organismos, coelhos e raposas. O sistema é constituído por uma matriz de n por m em que cada posição pode estar ou vazia ou ocupada por uma pedra, um coelho ou uma raposa.

A simulação corre num conjunto de gerações. Em cada geração inicialmente movem-se os coelhos e de seguida as raposas. Apenas um animal pode estar numa posição ao fim de uma geração.

Os animais podem reproduzir-se, deixando na sua posição anterior o novo animal.

As raposas podem morrer a fome ao fim de algum tempo, para tal não acontecer têm de comer um coelho, movendo-se para a sua posição.

1.1-Problema de paralelização

O objetivo foi criar uma implementação paralela eficiente para resolver o problema, com os algoritmos e estrutura deixada a cargo do aluno.

1.1.1-Algoritmo escolhido

Ao fim de várias tentativas e testes que serão descritos a detalhe nas próximas secções o algoritmo escolhido foi o seguinte.

Primeiro a matriz é lida do stdin de forma sequencial, sendo os objetos guardados na matriz e também em três listas, uma para cada tipo de objeto (pedra, coelho, raposa), com o objetivo de usar a lista para indexação.

Depois de lidos os dados é feito um ciclo, com cada iteração sendo uma geração. Dentro do ciclo procedesse inicialmente a mover os objetos.

As pedras são movidas primeiro, com um ciclo a iterar pela lista de pedras, estas são copiadas para a matriz da próxima geração.

De seguida os coelhos são movidos, com um ciclo a iterar pela lista de coelhos. Primeiro usando a matriz são encontradas as posições para que o coelho se pode mover, de seguida, e de acordo com as regras de movimento uma das posições é escolhida. Após uma posição valida ser escolhida, o coelho tenta procriar, deixando um novo na sua posição. Depois de procriar são resolvidos os conflitos entre coelhos a tentar mover-se para a mesma posição, apenas um sobrevivendo. Finalmente a matriz é modificada, o coelho passando a estar na nova posição.

Após os coelhos acabarem de ser movidos, as raposas são copiadas para a nova matriz e iniciam o seu movimento. O movimento das raposas é semelhante ao dos coelhos, tendo apenas duas diferenças. A primeira diferença é que as raposas primeiro tentam encontrar posições com coelhos para se moverem (tentando comê-los) e apenas se não houver coelhos adjacentes é que se tentam mover para posições vazias. A segunda é que uma vez que as raposas podem morrer á fome estas fazem-no apos procurar um coelho e não o encontrar.

Depois de todos os objetos agirem são feitas algumas operações de limpeza para preparar a próxima geração, nomeadamente reindexação das listas e reinício de matrizes auxiliares.

2-Implementação

O problema foi abordado de uma forma iterativa passando por algumas mudanças de estruturas de dados, organização da logica e implementação das áreas paralelas.

2.1-Estruturas de dados

Diversas estruturas de dados foram implementadas para organizar os dados.

Matrix (matriz)

Utilizada para guardar as matrizes. Uma vez que para o problema em questão todas as matrizes.

Várias funções relativas às matrizes foram criadas, para alocar, escrever, copiar e modificar todos os valores.

Na criação da matriz é alocada memoria contigua para os valores da matriz e um vetor de apontadores, com cada apontador a apontar para uma linha da matriz.

Object (objeto)

Usado para guardar informação sobre os objetos.

Como a posição, fome, procriação e tipo.

Várias funções relativas a objetos foram criadas, para alocar, escrever, copiar e modificar todos os valores.

List (lista)

Utilizada para indexar os dados e mais facilmente parti-los igualmente entre os diferentes threads. Em cada posição é guardado um objeto.

Passou por várias alterações.

Primeira versão:

Lista ligada clássica, com um apontador para o próximo elemento da lista.

Problemas:

Baixa eficiência a aceder a um elemento num índex em particular, e na remoção do mesmo.

Causa um aumento exponencial no tempo de execução com o aumento do número de objetos.

Segunda versão:

Vetor alocado com o tamanho total da matriz.

Problemas:

Utilização de memoria na maior parte dos casos muito superior ao necessário.

Terceira versão:

Vetor alocado com um tamanho inicial pequeno, quando não é suficiente um maior é alocado e o conteúdo anterior copiado. O tamanho alocado é aumentado exponencialmente, como tal não precisando de muitas realocações.

Apesar de algoritmicamente ser menos eficiente em termos de tempo, provou tem uma eficiência semelhante ou melhor que a versão anterior, simultaneamente ocupando muito menos espaço.

Várias funções relativas às listas foram criadas, para alocar, escrever, copiar e modificar todos os valores. Todas as versões têm as mesmas funções com os mesmos nomes e argumentos, permitindo alterar a utilizada apenas com a alteração do comando de compilação.

2.2-Leitura de dados

Os dados são lidos do stdin como indicado asseguir.

Na primeira linha encontram-se informações gerais sobre o programa e na seguinte ordem:

GEN_PROC_RABBITS – número de gerações até os coelhos poderem procriar
GEN_PROC_FOXES - número de gerações até as raposas poderem procriar
GEN_FOOD_FOXES - número de gerações até as raposas morrerem á fome
N_GEN - número de gerações na simulação
R - Número de linhas da matriz
C - Número de colunas da matriz
N - Número de objetos iniciais

A linha inicial é seguida de N linhas, uma com cada objeto no seguinte formato:

Objeto X Y

Com objeto sendo o tipo de objeto, por exemplo ROCK, X a coluna e Y a linha.

Por exemplo:

```
2 4 3 6 5 5 9
ROCK 0 0
RABBIT 0 2
FOX 0 4
FOX 1 0
FOX 1 4
ROCK 2 4
RABBIT 3 0
RABBIT 4 0
FOX 4 4
```

2.3-Divisão do trabalho

Como previamente mencionado, o movimento dos objetos é feito com 4 ciclos. Estes ciclos são do tipo “For”, e como tal permite facilmente dividir todos os objetos entre os diversos threads,

usando a operação “#pragma omp for”. No caso desta instrução vários tipos de *Schedule* foram testados, mas todos provaram ser equivalentes ou piores que *static (o tipo por defeito)*, como tal este foi mantido.

2.4-Sincronização

Para sincronizar as diversas threads, foi utilizado um vetor de locks. O acesso aos locks é feito com uma função de hash muito simples baseado na posição da matriz para que um objeto pretendo mover-se. A função é:

((número da coluna) + (número de linha) * (número de colunas)) modulo (tamanho do vetor)

Esta sincronização é bastante flexível, permitindo vários tipos de sincronização.

Foi implementada uma forma simples de escolher o tipo de sincronização, utilizando uma constante global chamada LOCKTYPE.

Tipos de lock:

CELL: Cada posição da matriz tem um único lock, apenas sendo preciso sincronizar quando dois objetos distintos tentam mover-se para a mesma posição em simultâneo. Para este tipo o tamanho do vetor de locks é definido para o tamanho da matriz.

ROW: Cada linha da matriz tem um único lock, apenas sendo preciso sincronizar quando dois objetos distintos tentam mover-se para a mesma linha em simultâneo. Para este tipo o tamanho do vetor de locks é definido para o número de linhas.

THREAD: Um lock por cada thread. Para este tipo o tamanho do vetor de locks é definido para o número de threads.

SINGLE: Um único lock, apenas um objeto se pode mover de cada vez. De todos o menos eficiente, no entanto, ocupa substancialmente menos espaço e tem menor tempo de inicialização.

FIXED: Um número fixo de locks. O número pode ser definido através da constante global DEFAULTLOCKS.

Os locks apenas foram bloqueados durante a resolução de conflitos entre vários objetos e durante a alteração dos valores de uma matriz após o movimento ser escolhido. Em todos os outros casos a execução é totalmente paralela, como por exemplo, na seleção do local para onde mover e na procriação (uma vez que o local está ocupado nenhum outro objeto tentara mover-se para lá).

Nas versões iniciais do programa as listas tinham um papel mais ativo para alem de indexação. Nesse caso durante a adição e remoção de elementos nas listas foram usadas zonas críticas.

2.5-Notas sobre nomenclatura

Durante todo o projeto uma nomenclatura consistente foi utilizada e é agora explicada para facilitar o entendimento do programa.

Todos os ficheiros estão em pascal case (PascalCase), tal como todos os structs.

Todas as funções estão em camel case (snake_case).

Todas as variáveis estão em camel case (camelCase).

Todas as constantes estão com tudo em maiúsculas (CONSTANT).

Foi adotada uma estrutura semelhante a linguagens orientadas a objetos, com uma struct principal correspondendo a um objeto. Cada struct tem um ficheiro para si com o mesmo node da struct. As funções que corresponderiam a métodos do objeto estão com o nome no formato StructName_function, por exemplo Matrix_print.

2.6-Notas sobre execução

Várias constantes foram definidas para facilmente controlar a execução do programa.

NTHREADS – Número de threads com que correr o programa.

NRUNS – Número de vezes que o programa será corrido para medição do tempo de execução.

LOCKTYPE – Tipo de lock a ser utilizado, tipos descritos na secção 2.4.

DEFAULTLOCKS – Número de locks a usar por defeito.

Existem 3 ficheiros .c e .h, os do List1, List2 e List3. List1 é a lista standard, List2 é o vetor de tamanho fixo e List3 é o vetor de tamanho dinâmico.

Existem também 2 ficheiros Project, o 2.1 e o 2.2. Sendo o 2.2 o trabalho final mais eficiente.

Fazendo o comando “make” o projeto final é compilado. Também é possível compilar 3 versões anteriores, com “make L1”, “make L2” e “make L3”. Cada um utiliza uma das listas, 1,2,3 respetivamente.

3-Desenvolvimento e análise de performance

Todos os tempos mencionados ao longo desta secção são a media de 10 medições.

Inicialmente o plano foi a utilização de matrizes para obter informação sobre adjacência de diversos objetos e listas para manipulação e divisão dos objetos.

A primeira lista utilizada foi uma lista ligada normal. No entanto rapidamente foram encontrados problemas sérios de performance á medida que o tamanho do problema aumenta. Tais problemas foram causados na tentativa de obtenção de um elemento específico da lista. Devido á natureza sequencial da lista era necessário percorrer toda a lista cada vez que se tentava obter um objeto. Devido a isto para a lista de tamanho 100 o programa tem o tempo de execução superior a meia hora.

Para tentar otimizar foram criadas duas versões alternativas de listas. Uma ocupando espaço constante ao longo da execução, enquanto outra que amenta o espaço apenas se necessário.

Para decidir entre ambas as listas foram feitos testes de performance nas matrizes de 100x100 e 200x200, obtendo 15.9143s, 63.4133s para a primeira e 15.6997s, 63.8996s para a segunda.

Uma vez que ambas são quase iguais em termos de tempo de execução, a segunda (que ocupa menos espaço) foi escolhida. A partir desse momento todo o desenvolvimento foi feito usando esta lista.

Na continuação do desenvolvimento outro problema de performance foi encontrado. Uma vez que as listas são atualizadas enquanto os objetos se movimentam, é necessário, no caso de conflitos entre objetos diferentes remover o que morre da lista, mas uma vez que não se tem o índice da posição é necessário percorrer toda a lista. Isto combinado que a lista tem de ser bloqueada enquanto se percorre, diminui bastante a performance e a capacidade de paralelização. Para corrigir o problema foi adicionado aos objetos um novo atributo, um valor que indica se está vivo ou morto. No caso de ter morrido é ignorado na geração seguinte. Com esta alteração foi possível melhorar o tempo nas matrizes de 100x100 e 200x200 para 12.8032s e 52.3022 respetivamente.

Apesar da melhoria, a existência de zonas críticas continuava a causar problemas de performance, como tal, uma forma alternativa foi testada. Invés de atualizar as listas durante o movimento, depois de este ocorrer percorrer toda a matriz e atualizar as listas. Com isto foi possível melhorar os tempos para 10.9391s e 44.5985s.

Agora que o programa funciona melhor com problemas de maior dimensão a atenção foi virada para o número de locks e para o *Schedule* dos ciclos.

Abaixo encontram-se os tempos de execução para diferentes tipos de lock e número de threads. Todos os tempos correspondem á matriz de 200x200.

	Número De Threads				
Tipo De Lock	1	2	4	8	16
Single	44.261s	63.645s	121.235s	181.330s	338.380s
THREAD	43.736s	60.426s	70.110s	72.036s	73.777s
FIXED: 10	43.756s	57.485s	62.594s	69.297s	93.002s
FIXED: 50	43.629s	50.573s	43.483s	44.886s	49.309s
ROW	44.246s	46.721s	36.669s	32.549s	35.426s
CELL	45.001s	44.618s	33.994s	29.277s	28.816s

Tabela 1: Tempo de computação para locks de diversos tipos e diversos números de threads (o valor é a media de 10 medições)

É possível observar que o tempo de execução diminui tanto com o aumento do número de locks como com o número de threads, como tal o tipo CELL será utilizado daqui para a frente.

Finalmente vários tipos *schedule* foram testados, mas todos provaram ser equivalentes ou piores que *static* (o tipo por defeito), como tal este foi mantido.

Uma vez que se tem todos os parâmetros definidos agora são apresentados os resultados finais.

	Número De Threads				
Tamanho	1	2	4	8	16
5x5	0ms	0ms	0ms	0ms	0ms
10x10	2ms	2ms	3ms	3ms	4ms
20x20	42ms	48ms	50ms	53ms	61ms
100x100	10.821s	10.648s	8.647s	7.311s	7.130s
100x100_u1	7.972s	7.875s	6.767s	6.196s	5.999s
100x100_u2	10.461s	10.432s	8.479s	7.235s	7.006s
200x200	45.205s	45.058s	34.730s	29.390s	28.601s

Tabela 1: Tempo de computação para matrizes de diversos tamanhos e diversos números de threads (o valor é a media de 10 medições)

	Número De Threads				
Tamanho	1	2	4	8	16
5x5	1	1	1	1	1
10x10	1	1	0.66	0.66	0.5
20x20	1	0.88	0.84	0.79	0.69
100x100	1	1.02	1.25	1.48	1.52
100x100_u1	1	1.01	1.18	1.29	1.33
100x100_u2	1	1	1.23	1.45	1.49
200x200	1	1	1.30	1.53	1.58

Tabela 2: Speedup para matrizes de diversos tamanhos e diversos números de threads (o valor é a media de 10 medições)

Ao aumentar o número de threads não se obteve uma melhora de performance tão boa quanto se esperaria. No entanto obteve-se um aumento suficientemente elevado para que a implementação paralela seja justificada.

4-Conclusão

Com este trabalho foi possível observar as vantagens do paralelismo, mas também as suas desvantagens. Se não for feito corretamente, não só não se tem um aumento de performance, mas também esta pode deteriorar-se. Também no caso em que a performance não é essencial, a dificuldade extra de implementação, assim como o tempo de o fazer tornam o paralelismo uma grande desvantagem. Em algumas situações pode não existir mais do que um processador, tornando o paralelismo irrelevante.