Assignment 2:
Ecosystem Simulation

Parallel Computing 2016/2017

Pedro Cardoso Belém
Nº 201602876
Rui Miguel Capela Fonseca
Nº 201602879

Base Flow

We decided to use two different worlds named “world” and “new_world”. The “world” will represent the original space before a set of movements and “new_world” will contain the new positions as they are being calculated. This way the objects already moved in a generation will not interfere with the ones still needing to calculate the new position. After this, both worlds are swapped, and the “new_world” is cleaned. Before the foxes are moved, the rabbits are copied to the new world so there aren’t any conflicts. So the normal flow of a generation is:

- “world” with the original positions, and “new_world” with no animals;
- The rabbits are moved, result in “new_world”;
- Swap worlds;
- Clean “new_world” and copy the rabbits;
- The foxes are moved, result in “new_world”;
- Swap worlds;
- Clean “new_world”.

We represent both worlds as an bidimensional array of structs which represent an object.

Parallelism

In most of this actions we iterate through the world, so those are fulcral points of parallelism. We use the base flow, but we divide the world so different threads could work in different sections, at the same time.
It’s also important that we have synchronization points after moving the rabbits, and at the end of a generation, to ensure that the foxes are moved first and that all objects are moved before starting a new generation.
To ensure that the manipulations in “new_world” are synchronized, we have a lock for each section of the map, depending on how we segmented the world.

We tried two variations:

- Parallel algorithm in which the work is distributed by rows. Each thread is responsible for different rows and each row has a lock associated with it.
- Parallel algorithm in which the work is distributed by sections which will have N/P size, such that N is the total size of the world and P is the number of threads that will execute. Each section has it’s own lock associated.

Results

- The results were measured on the given 5x5, 10x10, 20x20, 100x100. 200x200 inputs.
- The time is measure from after reading the input until it reaches the last generation.
- All the measures is the average of 5 runs.
- The parallel algorithms were tested with 1, 2, 4, 8 and 16 threads.

Sequential

Input size	Execution time (in milliseconds)
5x5	0.022
10x10	1.07
20x20	40.95
100x100	11823.60
200x200	44520.94

Parallel Row

White column - Execution time (in milliseconds)
Green column - Speed Up ratio
Orange column - Efficiency ratio

Ti	1			2			4		
5x5	0.048	1	1	0.09	0.53	0.27	0.116	0.41	0.10
10x10	1.60	1	1	2.02	0.79	0.40	2.39	0.79	0.20
20x20	50.09	1	1	48.07	1.04	0.52	48.87	1.02	0.26
100x100	12060.06	1	1	8114.23	1.49	0.75	5313.71	2.27	0.57
200x200	46855.13	1	1	31021.93	1.51	0.76	19400.76	2.42	0.61

Ti	8			16		
5x5	0.14	0.34	0.04	0.33	0.11	0.01
10x10	3.36	0.47	0.06	20.14	0.08	0.01
20x20	50.69	0.99	0.12	63.80	0.78	0.05
100x100	4236.18	2.85	0.36	5035.46	2.40	0.15
200x200	13612.77	3.44	0.43	12515.02	3.74	0.23

Parallel Grid

White column - Execution time (in milliseconds)
Green column - Speed Up ratio
Orange column - Efficiency ratio

Ti	1			2			4		
5x5	0.06	1	1	0.11	0.55	0.28	0.17	0.41	0.10
10x10	2.37	1	1	2.61	0.91	0.46	0.90	0.79	0.20
20x20	75.86	1	1	63.65	1.19	0.56	57.84	1.31	0.33
100x100	19882.97	1	1	14184.23	1.40	0.68	13140.23	1.5	0.38
200x200	79533.48	1	1	57117.64	1.39	0.68	55077.65	1.44	0.36

Ti	8			16		
5x5	0.17	0.34	0.04	0.34	0.55	0.03
10x10	3.82	0.47	0.06	5.94	0.40	0.03
20x20	65.69	0.99	0.12	98.93	0.77	0.05
100x100	13702.12	2.85	0.36	12511.8	1.59	0.10
200x200	59294.38	1.34	0.7	108608.9	0.73	0.05

Conclusions

- In overview, we can conclude that by increasing the input size, the speed up ratio is more noticeable, and in lower input sizes the cost of creating and destroying teams of threads can even outrank the parallelism gain. Efficiency also increases with input size.
- Even though we almost always get an increase in speed up by increasing the number of threads, we can see that this gain is minimal, observing the drop in efficiency.
- The version where we segmented the world in p (number of threads) segments isn't effective for two reasons:
    - High cost on calculating the index of the locks.
    - As each thread operates in more than a row, there is a lot of cache misses which will slow the execution time.