Rabbits and Foxes Ecosystem

Nuno Neto
up201703898

January 17, 2021

Parallel Computing
Masters in Computer Science

1 Base algorithm idea

The ecosystem functions in generations, each generation was implemented as follows:

1.1 Generation

To implement a single rabbits and foxes generation I used two main methods:

```
void performRabbitGeneration(int threadNumber,
    int genNumber, InputData *inputData,
    ThreadedData *threadedData,
    WorldSlot *world, WorldSlot *worldCopy,
    int startRow, int endRow;
void performFoxGeneration(int threadNumber,
    int genNumber, InputData *inputData,
    ThreadedData *threadedData,
    WorldSlot *world, WorldSlot *worldCopy,
    int startRow, int endRow);
```

Which handle moving the rabbits and the foxes, respectively. Rabbits are moved first, so that function will always be called before the foxes. Each of these methods receive, amongst other things needed for their operation, the *startRow* and *endRow* of it's territory. This is because I want to be able to only process a certain area of the ecosystem in preparation for the multi-threading. Because I are only processing a portion I only need a partial copy of the ecosystem for each thread to calculate the possible movements for their entities. I also know that I for movements that are within our area are never going to access another threads critical memory, as no 2 threads share the same area. This hoIver means that for movements that go away from a thread need to be sent to the correct thread for the movements to be processed, but for now I will focus on the sequential

code execution, where our functions have the full area.
I go over our area and when I find an entity, I will move it with the following methods:

```
void tickFox(int genNumber,int startRow,int endRow,
    int row,int col,WorldSlot *slot,
    InputData *inputData,WorldSlot *world,
    FoxMovements *foxMovements,Conflicts *conflictsForThread)
void tickRabbit(int genNumber,int startRow,int endRow,
    int row, int col,WorldSlot *slot,
    InputData *inputData,WorldSlot *world,
    RabbitMovements *rabbitMoves,Conflicts *conflictsForThread)
```

These methods handle the actual moving of each entity and adding to the conflict list if the move goes outside our ecosystem area. Entities can move to the north, Ist, south and east squares. They cannot move in a diagonal and they cannot move to places that have rocks or are larger than the ecosystem size.
Rabbits are moved first and they always move to an adjacent empty cell (If there are no empty cells I just don't move). After moving all the rabbits, I then move the Foxes. Foxes will prefer moving towards squares that have a rabbit but then can also move to an empty cell if or stand still if there is no possible movements.
Neither entity can move to a square where there is an entity of the same type at the time of moving. HoIver, more than one entity of the same kind can move to the same **empty square**. When this happens, only one of the entities can stay alive. The criteria for choosing what entity lives are:

For rabbits:

- Procreation age (Closest to procreate survives)

For foxes:

- Procreation age (Closest to procreate survives)
- When they have the same procreation age, the tie breaker is the Food age (Furthest from death survives)

There is no tie breaker on rabbits so I just kept the rabbit that was already at the position.

1.2 Running the several generations

To run these generations I will have to call the methods described above repeatedly until I reach our goal generation count.

To implement performing a single generation I created the following method:

```
void performGeneration(int threadNumber,int genNumber,
                       InputData *inputData,ThreadedData *threadedData,
                       WorldSlot *world,
                       ThreadRowData *threadRowData)
```

This method will call the *performRabbitMoves()* and *performFoxMoves()*. It will also handle making the copies of world that are needed and synchronizing where it's needed.

To execute a large amount of generations I created the following method:

```
void executeWithThreadCount(int threadCount,
    FILE *inputFile,FILE *outputFile)
```

This method will handle creating all the threads, reading the input data and writing the outputs. It also iterates the *performGeneration* method as many times as needed with a for.

1.3 Data Structures

The main data structures are:

```
typedef struct InputData_ {
    int gen_proc_rabbits;
    int gen_proc_foxes, gen_food_foxes;
    int n_gen;
    int rows, columns;
    int initialPopulation;
    int threads;
    int rocks;
    int *entitiesAccumulatedPerRow;
    int *entitiesPerRow;
} InputData;
```

Here I store the default information of the ecosystem and the entity count both per row and per row + all the rows that come before it. This is used to dynamically assign the areas of the ecosystem according to the entity count. This is done to avoid having threads that are processing a large portion of the work because they have a very large entity count in their area, while other threads have areas that are almost empty and therefore don't have a lot of work to do. This will be discussed further, in Section 4.2.

```
typedef struct RabbitInfo_ {
    int genUpdated, prevGen;
    int currentGen;
} RabbitInfo;
typedef struct FoxInfo_ {
    int genUpdated, prevGenProc;
    int currentGenProc;
    int currentGenFood;
} FoxInfo;
```

These two structs store the information for each rabbit and fox, respectively. Each allocated instance of these structs represent a living entity and freeing the object means that the entity has died.

```
typedef enum SlotContent_ {
    EMPTY = 0,
    ROCK = 1,
    RABBIT = 2,
    FOX = 3
} SlotContent;
```

Here we store the possible states of an ecosystem position.

```
typedef struct WorldSlot_ {
    SlotContent slotContent;
    int defaultP;
    MoveDirection *defaultPossibleMoveDirections;
    union {
        FoxInfo *foxInfo;
        RabbitInfo *rabbitInfo;
    } entityInfo;
} WorldSlot;
```

This is the struct that represents a single ecosystem position. Each position stores its contents and it's default movements. Storing the default movements allows us to not have to search each

direction when there are some directions that will never be possible (Into a rock or outside the ecosystem area). This is still memory efficient as all slots that have all possible moves share a *MoveDirection** instance. *defaultP* is where the amount of default possible movements is stored.

2 Division of work

The division work was done dynamically for each generation and the ecosystem was divided by rows. The entity counts in the struct InputData are updated on each generation automatically by the algorithm with no extra cost to the performance (Just incrementing numbers in an array). This didn't have to be synchronized as I divide the work by rows so each thread will only increment the entity count of rows in their area.
Using this information, I calculate the accumulated entity count for each thread (synchronized in ascending thread order to make sure the numbers are correct).In the last thread I then search for the average of the entities per thread, which is done by dividing the total entity count by the number of threads, multiplied by the thread number. This gives a target to search for and I then employ a binary search for each thread. This leads to a complexity of $\mathcal{O}(t\ log_2(r))$, where t is the thread count and r is the number of rows. After this is calculated, all threads start their work, which is in theory now divided equally amongst all the threads.

2.1 Synchronization points

I synchronize among threads after each rabbit generation to handle the rabbit conflicts (Here I only have to synchronize with the threads that are above and below ours), after handling the rabbit conflicts to make sure all threads have an up-to-date world copy, after that I have to synchronize after the foxes have been moved to handle the fox conflics (Much like with the rabbits, I only have to wait for thread above and bellow ours) and after we have handled all the conflicts we synchronize in order to perform the dynamic allocation. I also synchronize after each world copy to make sure some threads don't start changing the world before all the threads have their copy ready.

2.2 Data structures

Here are the main data structures used for the division of work:

```
struct ThreadedData {
    Conflicts **conflictPerThreads;
    pthread_t *threads;
    sem_t *threadSemaphores, *precedingSemaphores;
    pthread_barrier_t barrier;
};
```

This struct stored all of the pthreads objects that are used to control access to critical areas along with all of the conflicts for each thread.

```
typedef struct Conflict_ {
    int newRow, newCol;
    SlotContent slotContent;
    void *data;
} Conflict;
typedef struct Conflicts_ {
    int aboveCount;
    Conflict *above;
    int bellowCount;
    Conflict *bellow;
} Conflicts;
```

These structs store the movements that are made by the surrounding threads. At the end of each rabbit and fox movements we will synchronize with the surrounding threads and process these movements.

```
typedef struct ThreadRowData_ {
    int startRow, endRow;
} ThreadRowData;
```

This struct is used to store each thread's ecosystem area.

3 Performance

	Times and speedups										
Input:	Sequencial	1 Thread	Speedup	2	Speedup	4	Speedup	8	Speedup	16	Speedup
5x5	0,003	0,003	1	0,004	0,75	0,004	0,75	DNR	0	DNR	0
10x10	0,005	0,005	1	0,009	0,6	0,01	0,5	0,015	0,3	DNR	0
20x20	0,049	0,048	1	0,089	0,6	0,088	0,6	0,11	0,44	0,212	0,2
100x100	11,408	11,41	0,9	6,969	1,7	4,23	2,7	3,03	3,8	2,985	3,82
100x100 Unbal 01	7,959	8,085	0,98	5,232	1,5	3,274	2,4	2,508	3,2	2,537	3,14
100x100 Unbal02	10,883	11,0023	0,99	6,937	1,6	4,157	2,6	3,008	3,6	2,942	3,7
200x200	48,187	48,938	0,98	25,808	1,87	13,7	3,5	7,772	6,2	5,95	8,1

As it's possible to see, the performance with 1 thread is a little worse than sequential, as one would have expected given the overhead of starting the threads and other thread related work.

When we move to 2 threads, we see a good increase in performance when we start getting to larger input sizes, with a maximum speedup of almost 1.9.

Moving to 4 threads we again see the best improvement in the largest input sizes with a 3.5x speedup for the input size 200x200. We see pretty similar speedups in the unbalanced as the balanced 100x100 input, which tells us that the dynamic allocation is doing it's job.

Moving to 8 and 16 threads we see a repetition of the story, where there's a small increase for the smaller input sizes (100x100) and a decent increase for the larger input 200x200. Where we see a 6.2 speedup for 8 threads and a 8.1 speedup for 16 threads. We also see that going from 8 to 16 threads doesn't produce that much better of a speedup, this is likely because the 200x200 input isn't really large enough to compensate the time that the threads have to spend synchronizing.

4 Other explored alternatives

Some other alternatives were explored, some even came out with better speedups. However when I tested them I still did not have access to the machine and the speedups on my local machine were worse or as good at best so at the time I decided not to explore them further. When I noticed that they did perform better it was already too late to implement the dynamic allocation, so I had to deliver the alternative that I had already developed. However with more time I believe I could have gotten better results. These alternatives may not produce the correct output as the only one I corrected all the bugs on was the one I delivered, again because of time.

4.1 No middle copy

In this alternative I found a way to not perform a copy of the world between the rabbits and the foxes generation, which reduced the amount of synchronizing required. This produced the following results:

	Times and Speedups								
Input:	Sequencial (1 Thread)	2	Speedup	4	Speedup	8	Speedup	16	Speedup
5x5	0,003	0,004	0,75	0,006	0,5	DNR	0	DNR	0
10x10	0,006	0,011	0,6	0,012	0,5	0,017	0,4	DNR	0
20x20	0,051	0,076	0,7	0,081	0,6	0,115	0,4	0,175	0,3
100x100	12,331	7,672	1,6	4,458	2,8	2,712	4,6	2,49	5
100x100 Unbal 01	8,626	5,567	1,5	3,353	2,6	2,263	3,8	2,306	3,7
100x100 Unbal02	11,951	7,429	1,6	4,299	2,8	2,675	4,4	2,503	4,8
200x200	54,714	28,104	1,95	15,229	3,6	8,233	6,7	5,314	10,3

As we can see, there are some speedups that could greatly benefic from dynamic allocation, but in general, we see some better speed ups compared to our delivered attempt.

4.2 No middle synchronization

In this other alternative I found a way to completly remove the middle synchronization from the generations and only synchronize in the end of the thread. This was done by simulating the moves of the rabbits that are not in our control but that still can affect the moves of the rabbits and foxes in our area. The 2 lines above and 2 lines bellow were simulated and that information was used when moving the foxes. At the end of the generation, threads synchronize with the threads above and bellow it and solve the rabbits and foxes conflicts. These were the results:

	Tempos								
Input:	Sequencial (1 Thread)	2	Speedup	4	Speedup	8	Speedup	16	Speedup
5x5	0,003	0,004	0,75	0,004	0,75	DNR	0	DNR	0
10x10	0,005	0,009	0,6	0,01	0,5	0,011	0,5	DNR	0
20x20	0,048	0,063	0,8	0,065	0,74	0,067	0,72	0,082	0,6
100x100	11,522	7,298	1,57	4,425	2,6	3,045	3,8	2,696	4,3
100x100 Unbal 01	8,101	5,116	1,58	3,216	2,5	2,351	3,45	2,039	4
100x100 Unbal02	11,25	6,87	1,64	4,394	2,56	3,034	3,7	2,554	4,4
200x200	54,3	27,686	1,96	14,267	3,8	8,936	6,1	6,142	8,8

As we can see, there are some pretty good speed ups that could have been improved by the dynamic allocation, but again, when I tried these methods locally I saw little to no performance gain and therefore thought that it wasn't worth pursuing.