

**CIS6007 Parallel and Distributed Systems**

**Assignment B**

**By**

**Name: Filip Svetlomirov Petkov**

**Student ID: ST20283945**

**Programme: BSc (Hons) Software Engineering**

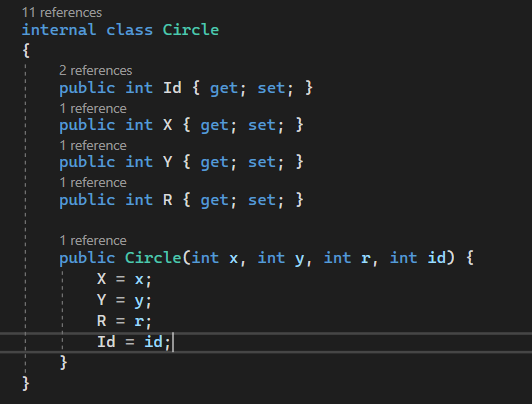
**Github repository link:**

<https://github.com/Hellsfan/Parallel_And_Distributed_Systems_Assignment_B.git>

**Task 1. Painting Circles in Parallel with multiple Threads**

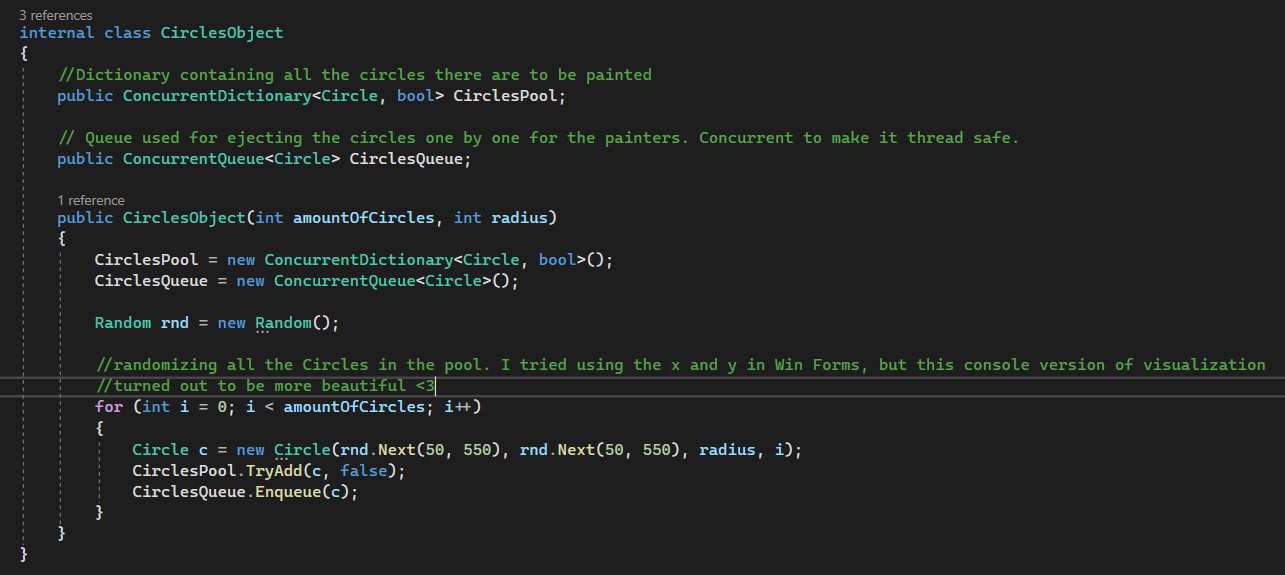
**Part 1.** Structure of the code:

First let us begin with the objects used for the code. First we have the circle object:



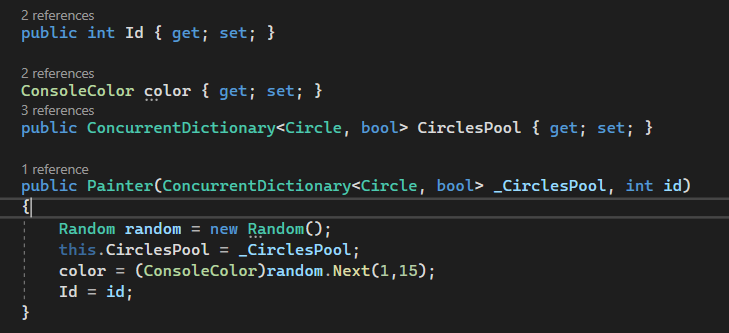
Fairly simple object, where all we need are Id, coordinates and radius. Then inside the constructor we set all those properties to the ones we want.

Then we have a Circles Object, which is used to generate a large pool of circles that will be used to work with.



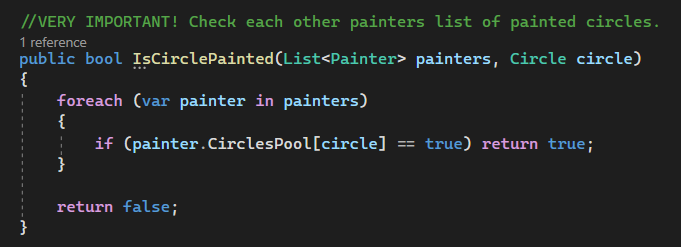
Here we have Concurrent Dictionary and Concurrent Queue which are used for making thread safe operations with them. Then in the constructor we generate the wanted amount of circles based on the random object for properties. Afterwards we add each circle to the dictionary and the queue. The reason why I used a dictionary instead of a list is for the speed at which access to a specific circle happens. With list the speed is O(n) where n is the amount of all circles. While with a dictionary it is O(1), due to the circle we want to check is the key for the dictionary and access is instant. Of course all circles start off as NOT painted, therefore all values are false in the dictionary.

Then we have the painter class:



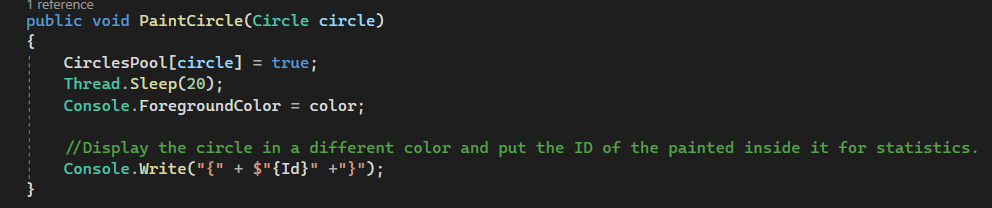
Who has Id for indexing, ConsoleColor for easier visualization and its own dictionary which is the same one we create from the Circles object. The console color is chosen at random from the already existing enum ConsoleColor.

Then we have the method which checks if a circle is already painted by another worker.



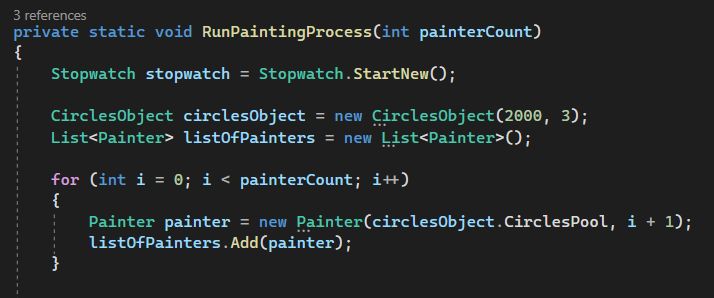
Now from the task requirements, workers cannot see an universal list of all circles and which ones are painted. Therefore I made each worker have to check each of the other workers dictionaries for a said circle, whether it is painted or not.

Afterwards we have the method which paints the circle:

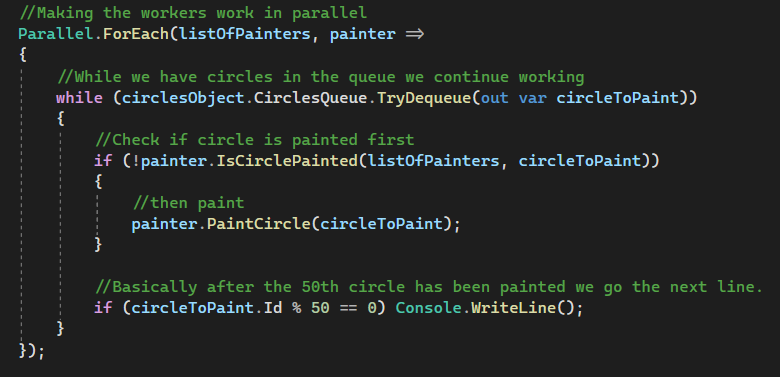


First we set the value in the workers pool as true, so that other workers know it is painted when they check the same circle. After that we set a delay of 20 msecs and change the foreground color of the console for easier visualization and display the circle in the console. Then the circle is considered as painted and we continue forth.

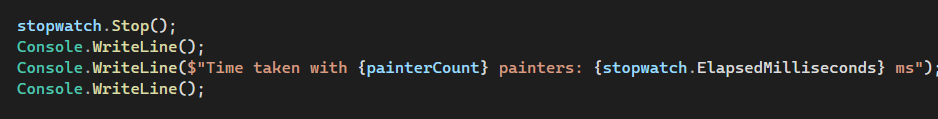
Now comes the part of the main logic of the code, the program.cs file:



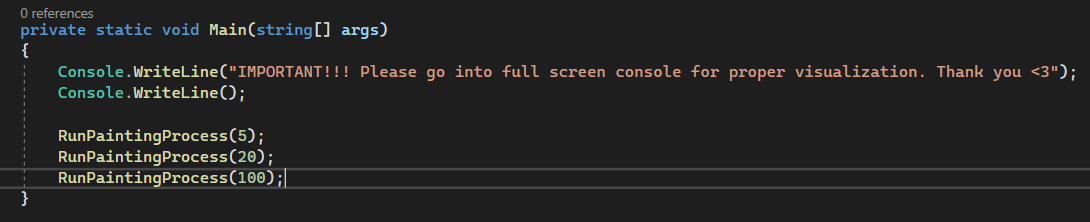
I made a function which accepts wanted painter count as a parameter and creates a stopwatch for counting the time. Afterwards we create a circles object with 2000 circles which have a radius of 3 pixels. We then make an empty list of painters and go into a small for loop which fills the empty list with different painters based on the given parameter.



I used Parallel.ForEach to make a separate thread for each worker and make the structure of the working algorithm. While we have objects in the queue we do not stop working. Then each painter asynchronously checks if a circle is painted and if said circle is not painted, starts painting it. Then for display purposes after each 50th circle is painted we start displaying the circles on the next line. Normally we display them with the Console.Write() method.

At the end we stop the stopwatch and display the recorded time in milliseconds.

Finally we call the function in the main method 3 times, all with different amount of workers: 5, 20, 100 to compare different execution times.



**Part 2. Evaluation of the task:**

**Is the problem able to be parallelized?**

Yes. The problem can be parallelized. Painting circles can happen completely independently, as well as checking which circle is painted. The only important thing is to manage access to the resources properly, due to possible race conditions.

**How would the problem be partitioned?**

In my approach, technically there is no partitioning. All Workers start from the beginning of the queue. Workers then receive a circle from the queue and check whether it's painted or not. Afterwards it is based entirely on the processing speed of the threads, where workers “race” each other for the next circle from the queue, until all the circles are painted. But I see a possible solution as well where instead of a queue, we can use a list and all workers start from different indexes of the list, therefore the list being partitioned between them.

**Are communications needed?**

In my solution, most certainly needed. Everytime a circle is given, the worker needs to perform a check to see if someone hasn’t already painted it yet. Also with the fact that circles are constantly being done and the dictionaries of workers constantly being updated it is very important to manage the communication properly and what time it happens, due to possible issues and race conditions.

**Are there any data dependencies?**

Yes. All workers depend on other workers' dictionaries. Due to the requirement that there is no universal way for all workers to check whether a circle is painted or not, all communication needs to happen between workers, therefore all of their data needs to be managed, especially when to update and when to read from it.

**Are there synchronization needs?**

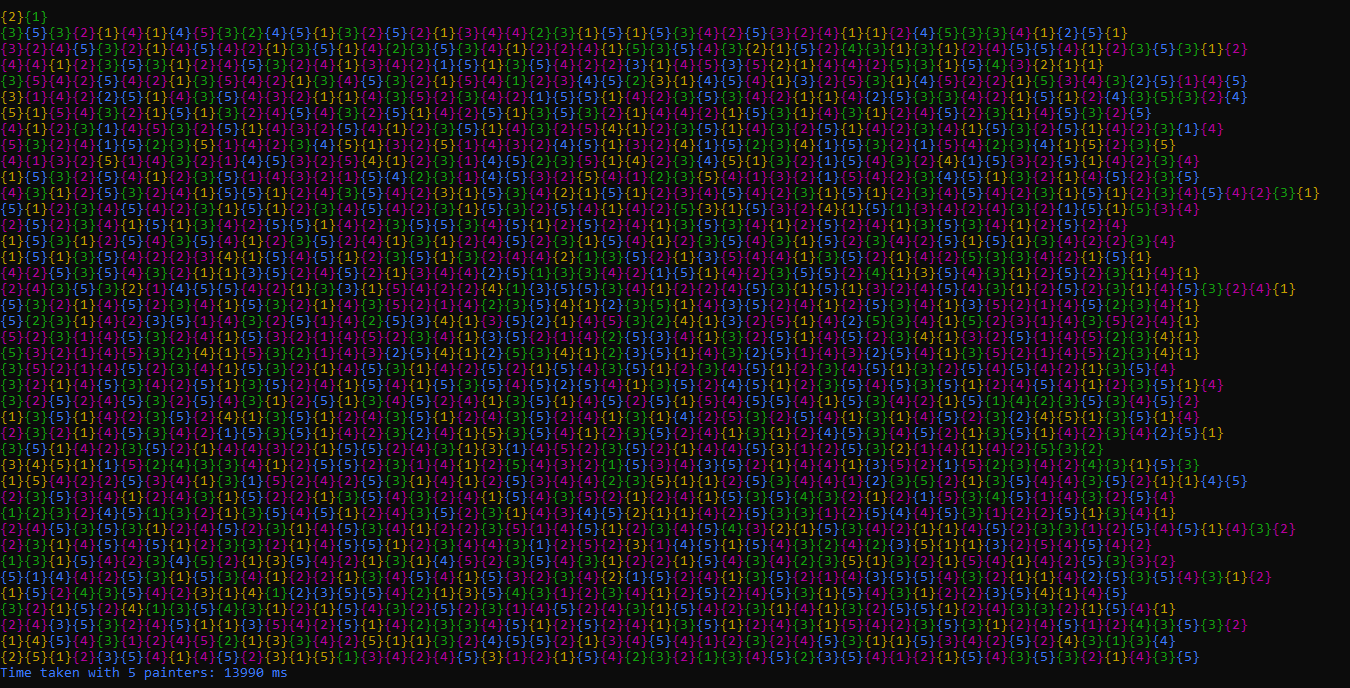
Yes. Reading and writing in a dictionary of circles at the same time is very dangerous. So we need to synchronize when all that happens. We need clear separation when other workers can read from a dictionary and when a worker writes in their dictionary. Both actions cannot happen at the same time. Assuming the resources are managed properly, painting circles is no issue, due to the painting itself to happen completely independent of any data.

**Will load balancing be an issue?**

In my solution, load balancing is all dependent on which thread receives priority from the CPU itself. All threads start from the very beginning of the queue, so whichever has high computational resources, gets to do more work. Therefore I believe it is safe to say that load balancing would not be an issue here.

**Part 3. Test results**

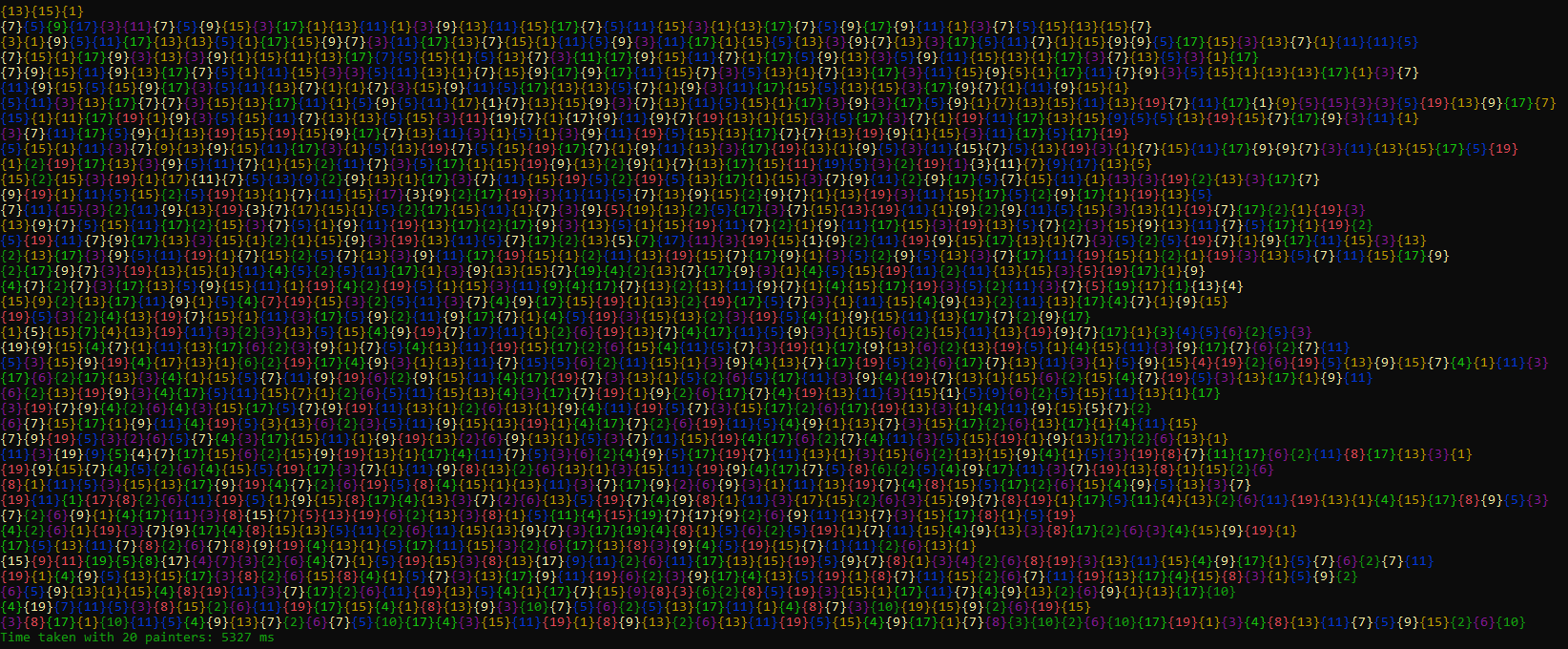
First we have a test with 5 workers with a circle pool of 2000 circles:





As we can see the execution time is about 14 seconds. The workload is distributed quite equally. The way I made it go to the next line is when the 50th circle is painted ( for example circle 50, circle 150, circle 1550 etc.). This shows that the computational speed between threads is almost identical and that a matter of 5-6 circles are painted more or less between splits.

Then we have a test with 20 workers:





Test with 20 workers improves the performance by a lot. The total time taken is about 5 seconds which more than double in contrast with 5 workers. In this case we see that between lines more circles were painted before the 50th element was hit (circa 9-10 circles more or less). Workload is also quite nicely balanced between threads judging by the ids of threads inside the circles.

Finally we have a test with 100 workers:





Now with 100 threads we see an improvement in performance as well. Around 3.6 seconds for 2000 circles. This shows that, although there are 5 times more workers in contrast with 20 workers, the performance gained is not 5 times more, but rather less than double. So we can conclude that there is a certain limit we can reach, where more workers does not equal the same performance gain. Hitting the 50th in order to split lines also seems to be wilder than before. Workload seems to be balanced here as well.

**Conclusion:**

In conclusion the task can be parallelized quite well and we can see improvement in performance throughout different amounts of workers. Clearly there is a possible golden ratio between the amount of workers utilized and performance increase, where afterwards resources spent does not seem to benefit us enough.