*SECTION B* – Documentation

*Module*: CIS6007 Parallel and Distributed Systems

*School*: Cardiff School of Technologies/Varna University of Management

Lecturer: IVELIN GEORGIEV

*Student number*: 20283946

Git hub link -> https://github.com/lightonray/st20283946\_CIS6007\_B

***STRUCTURE OF THE APPLICATION***

The provided C# is an application developed on Windows Forms that utilizes multithreading to concurrently paint circles on a graphical interface.

1. **Namespaces:**
   * Importing necessary namespaces for the application.
2. **Class Form1:**
   * Inherits from **Form** and represents the main form of the application.
   * Contains UI components and event handlers.
3. **Fields and Variables:**
   * **lockObject**: Object for synchronization purposes.
   * Constants for circle dimensions (**circleRadius**, **circleSpacing**, **maxCirclesPerRow**).
   * Variables (**numberOfCircles**, **numberOfWorkers**, **circlePaintingTime**) for controlling the circle painting process.
   * **paintedIndices**: HashSet to store indices of painted circles.
4. **Constructor (Form1()):**
   * Initializes UI components and sets default values for lists (**listBox1**, **listBox2**).
5. **Methods:**
   * **PaintCircle**: Simulates the painting of a circle by a worker thread, updating progress and drawing the circle on the UI.
   * **DrawCircle**: Draws a circle on the **PictureBox** with specified parameters.
   * **GetThreadColor**: Assigns a unique color to each worker thread.
   * **UpdateProgress**, **UpdateListBox3**: Methods to update progress information and UI elements safely.
   * **ParallelPaintCircles**: Orchestrates parallel painting of circles by multiple threads.
   * **btnStartPainting\_Click\_1**: Event handler for the "Start Painting" button, initiates circle painting.
6. **Threading Logic:**
   * **ParallelPaintCircles** method divides the circle painting task among multiple worker threads, manages their execution, and synchronizes their completion.
7. **UI Interaction and Update Handling:**
   * Interaction with UI elements such as progress lists (**listBoxProgress**, **listBox3**) and handling cross-thread UI updates using **Invoke**.
8. **Color Selection Logic:**
   * The **GetThreadColor** method assigns unique colors to worker threads based on their IDs.
9. **Simulated Painting Logic:**
   * The **PaintCircle** method simulates painting circles on the **PictureBox**, managing synchronization and UI updates within a multi-threaded environment.

This structural overview highlights the main components, methods, and functionalities present in the program, showcasing its organization and flow.

***FLOW OF THE APPLICATION***

1. Initialization:
   * Application initializes namespaces, UI components, and variables.
2. User Interaction:
   * User triggers the painting process by clicking the "Start Painting" button.
3. Threaded Circle Painting Process Begins:
   * ParallelPaintCircles method manages concurrent circle painting using multiple threads.
4. Worker Thread Creation and Workload Assignment:
   * ParallelPaintCircles divides total circles among worker threads.
   * Each worker thread is created, assigned a range of circles to paint, and starts executing PaintCircle.
5. Individual Circle Painting by Worker Threads:
   * PaintCircle simulates painting:
     + Assigns a unique color to the worker thread.
     + Updates progress, determines circle coordinates, and draws the circle on the PictureBox.
     + Handling Already Painted Circles:
       - Before painting a circle, it checks the paintedIndices HashSet to ensure the circle hasn’t been painted yet.
       - Synchronization with lock:
         * It uses a lock to synchronize access to paintedIndices.
         * Ensures that while one thread is checking or updating paintedIndices, other threads are blocked from modifying it.
       - If the circle has not been painted (!paintedIndices.Contains(circleId)), the thread proceeds with painting and updates the paintedIndices set.
6. Thread Synchronization and Completion:
   * Worker threads complete their assigned circles, synchronize, and finish their tasks.
7. Elapsed Time Calculation and Display:
   * Application calculates elapsed time for circle painting and updates UI with progress information.
8. UI Update Handling:
   * UI is updated with progress information using thread-safe methods (Invoke for cross-thread UI updates).

***Evaluation of the tasks according to the following criteria***

1. Parallelization Potential:
   * This problem can be effectively parallelized. Each circle painting operation is independent and doesn't rely on the results of other circles. Hence, multiple threads can concurrently paint circles without dependency issues.
2. Problem Partitioning:
   * The problem can be partitioned by assigning each thread a specific range of circles to paint. Threads work on different sets of circles, avoiding conflicts.
3. Communications Needed:
   * Minimal communication is required among threads. However, synchronization is necessary to avoid multiple threads attempting to paint the same circle simultaneously.
4. Data Dependencies:
   * Data dependencies exist concerning the shared information about painted circles (paintedIndices). Threads need to synchronize their access to this data to prevent conflicts.
5. Synchronization Needs:
   * Synchronization is crucial to ensure mutual exclusion when accessing the shared data (paintedIndices). Locks are used to prevent multiple threads from accessing or modifying the same data concurrently.
6. Load Balancing Concerns:
   * Load balancing might be a concern if the distribution of circles among threads is uneven. If some threads finish their work earlier than others due to an unequal workload, it can impact overall efficiency. Load balancing strategies might be necessary to distribute the workload evenly among threads.

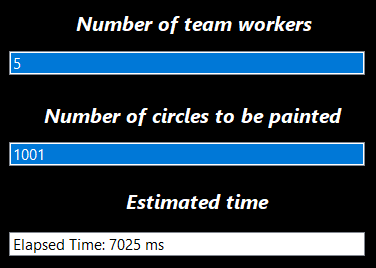
***TEST RESULTS***

***K (TEAM OF WORKERS) => 5***

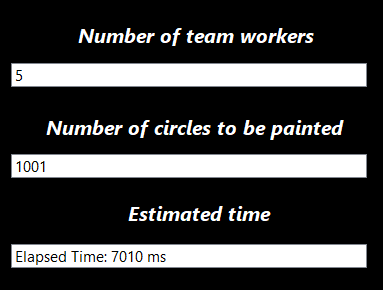
***DELAY PER PAINTING CIRCLE => 20 MSEC***

***NUMBER OF CIRCLES => 1001***

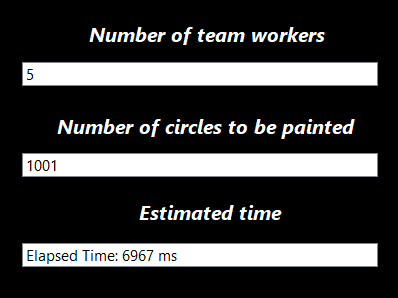
***1 Iteration => 7025ms***

******

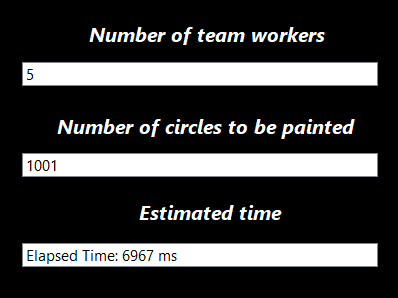
***2 Iteration => 7010ms***

******

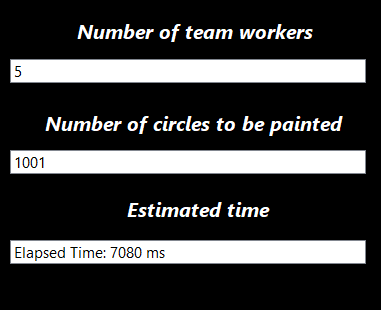
***3 Iteration => 6967ms***

******

***4 Iteration => 6967ms***

******

***5 Iteration => 7080***

******

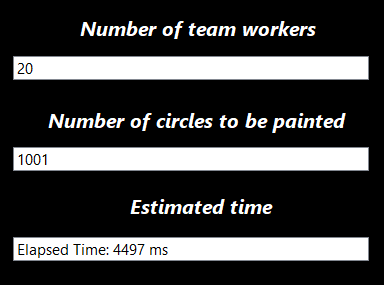
***AVERAGE FOR K = 5 => 7025ms + 7010ms + 6967ms + 6967ms + 7080ms = 7009,8 ms***

***K (TEAM OF WORKERS) => 20***

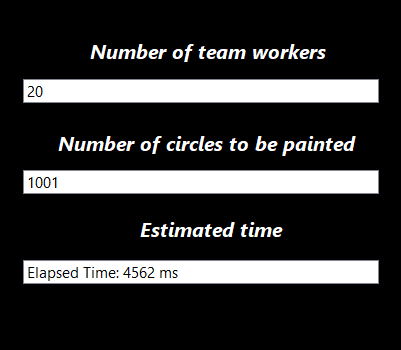
***DELAY PER PAINTING CIRCLE => 20 MSEC***

***NUMBER OF CIRCLES => 1001***

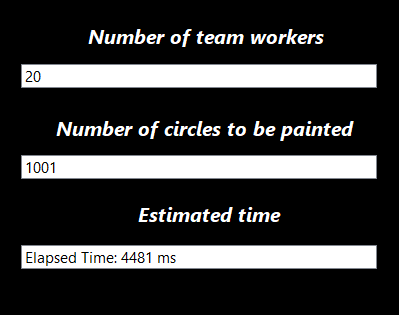
***1 Iteration => 4497ms***



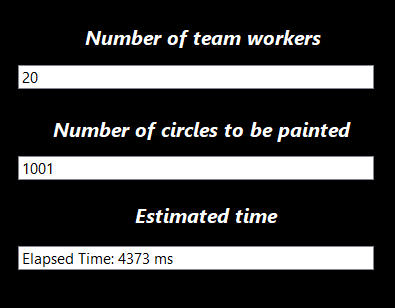
***2 Iteration => 4562ms***



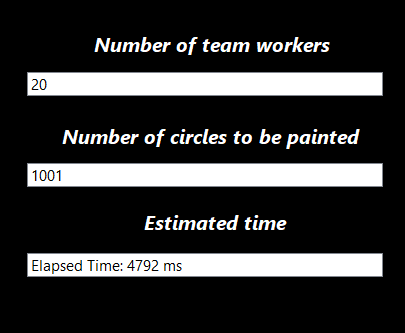
***3 Iteration => 4481ms***



***4 Iteration => 4373ms***



***5 Iteration => 4792ms***



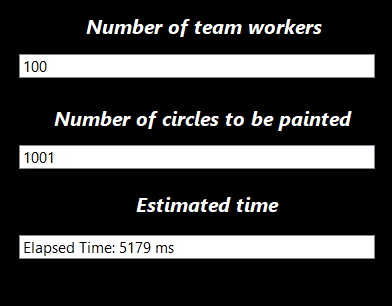
***AVERAGE FOR K = 20 => 4497ms + 4562ms + 4481ms + 4373ms + 4792ms = 4541ms***

***K (TEAM OF WORKERS) => 100***

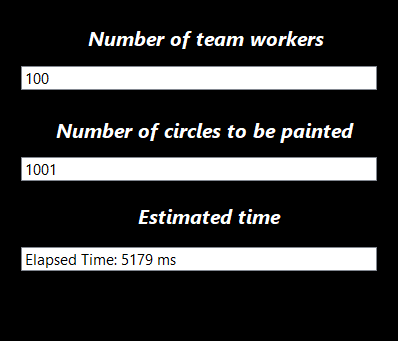
***DELAY PER PAINTING CIRCLE => 20 MSEC***

***NUMBER OF CIRCLES => 1001***

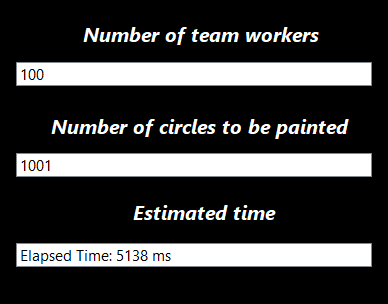
***1 Iteration => 5179ms***



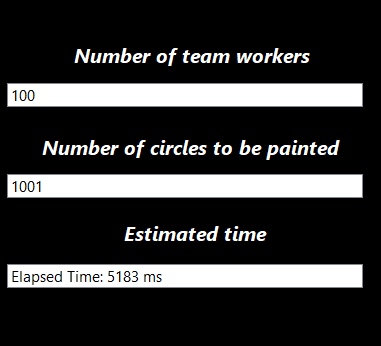
***2 Iteration => 5179ms***



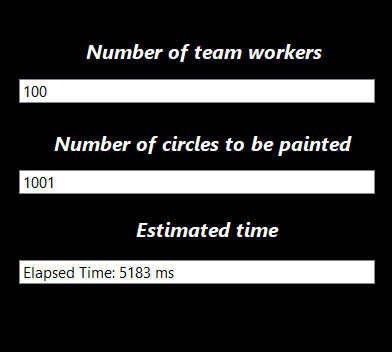
***3 Iteration => 5138ms***



***4 Iteration => 5183ms***



***5 Iteration => 5183ms***



***AVERAGE FOR K = 100 => 5179ms + 5179ms + 5138ms + 5183ms + 5183ms + 5183ms = 5174ms***

1. ***SUMMARY***5 threads:
   * Average for K=5 is approximately 7009.8 milliseconds.
   * This average indicates the average execution time when the system or computation is handling five concurrent threads. It suggests that with five threads, the overall processing time tends to hover around 7009.8 milliseconds on average. This could be due to factors such as thread synchronization, resource contention, or the nature of the workload.
2. 20 threads:
   * Average for K=20 is 4541 milliseconds.
   * This average represents the average execution time when the system is dealing with twenty concurrent threads. It indicates that with an increased number of threads (20 in this case), the overall execution time decreases to around 4541 milliseconds on average. This reduction might be due to better parallel processing or optimized resource allocation.
3. 100 threads:
   * Average for K=100 is 5174 milliseconds.
   * Here, the average execution time with 100 concurrent threads is approximately 5174 milliseconds. This shows that with an even larger number of threads (100 threads), the overall execution time slightly increases compared to the case of 20 threads. This increase could be due to potential overheads associated with managing a larger number of threads or resource limitations.