# CIS6007 Parallel and Distributed Systems Assignment 1 Name: Yumit Demir Student Id: 2218012 Program: SE3 Github repository link: [yumitdemir/CIS6007-Assignment-1 (github.com)](https://github.com/yumitdemir/CIS6007-Assignment-1)

Task 1

Parallel bubble sort is a variation of the bubble sort algorithm where the array is divided into smaller parts, allowing each part to be sorted simultaneously by multiple threads. Just like in standard bubble sort, two items are compared and swapped if necessary, repeating until no more swaps are needed. This parallel approach speeds up the process by taking advantage of multiple threads working together, and it can be implemented efficiently without significant synchronization issues.

## Code Structure:

### BubbleSortParallel Class:

* **GenerateRandomArray Method:** Generates an array of random integers.
* **ParallelBubbleSort Method:** Divides the array into segments, sorts each segment in parallel using bubble sort, and then merges the sorted segments.
* **BubbleSort Method:** Implements the bubble sort algorithm for sorting an array segment.
* **MergedArrays Method:** Merges multiple sorted segments into a single sorted array.

### Program.cs:

* Generates a random array.
* Iterates over different thread counts to perform parallel bubble sort.
* Measures and prints the time taken for sorting with each thread count.

## Evaluation of Task 1

### 1. Is this problem able to be parallelized?

Yes, the problem can be parallelized by using a batch processing approach. Each thread is assigned its own portion of the workload, and we can manage the threads through a shared criteria dictionary. This ensures that each thread operates independently while still adhering to the necessary sorting criteria, resulting in a more efficient sorting process.

### 2. How would the problem be partitioned?

The problem can be partitioned by dividing the original array into N segments, where N is the number of threads. Each thread independently sorts its assigned segment using the bubble sort algorithm. After sorting, the threads merge the sorted segments into a single sorted array. Proper synchronization is essential during the merging phase to prevent race conditions.

### 3. Are communications needed?

Yes, communication is needed during the merging phase, where the sorted segments from each thread must be combined into a single sorted array.

### 4. Are there any data dependencies?

In parallel bubble sort, data dependencies primarily arise during the merging phase, where the sorted segments must be combined into a single array. Each thread sorts its segment independently, so there are no data dependencies during sorting. However, the merging process requires access to the sorted segments, which necessitates coordination to ensure the correct order and avoid race conditions.

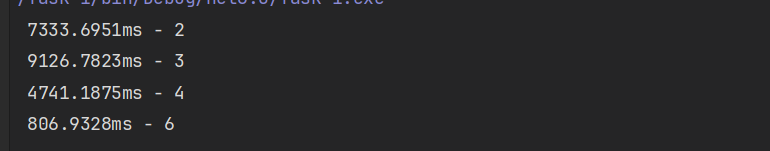
### 5. Are there synchronization needs?

Yes, synchronization is essential to prevent data corruption. A locking mechanism ensures that threads do not overwrite the criteria dictionary simultaneously, which could lead to inconsistencies in the sorting process.

### 6. Will load balancing be a concern?

Yes, load balancing is a concern in parallel bubble sort. If segments are unevenly distributed among threads, some threads may finish their sorting tasks faster than others, leading to idle resources and inefficient execution. To optimize performance, it's important to ensure that each thread receives an approximately equal workload.

## Task results and observations



The test results show that sorting time decreases significantly as the number of threads increases, with the best performance at 6 threads (806.9328 ms). However, using 3 threads (9126.7823 ms) is slower than 2 threads (7333.6951 ms), indicating potential overhead from managing more threads. Overall, the data suggests that parallel bubble sort is effective, but optimal thread count should be carefully considered to maximize performance.

Task 2

## Code Structure:

### Inventory Class:

* **GenerateInventory Method:** Generates a list of items with random types, barcodes, and descriptions.

### Item Class:

* Represents an item with properties for type, barcode, and description.

### ParallelSearch Class:

* **SearchInventory Method:** Searches the inventory in parallel to find items matching the target counts for each type.
* **InitializeFoundItems Method:** Initializes a dictionary to store found items.
* **IsSearchComplete Method:** Checks if the search is complete based on the target counts.

### Program.cs:

* Generates an inventory list.
* Defines target counts for item types.
* Iterates over different thread counts to perform parallel search.
* Measures and prints the average time taken for searching with each thread count.

## Evaluation of Task 2

### 1. Is this problem able to be parallelized?

Yes, this problem can be parallelized because the search tasks for different item types are independent of each other. By using multiple threads, the SearchInventory method can perform searches concurrently, significantly improving performance, especially with larger inventories. This approach leverages CPU resources effectively, making it well-suited for parallel execution.

### 2. How would the problem be partitioned?

The problem can be partitioned by dividing the inventory based on item types, with each thread responsible for searching a specific type or a chunk of the inventory. This allows for efficient parallel processing and reduces contention among threads.

### 3. Are communications needed?

That's correct! Minimal communication is needed since threads operate independently, but they update a shared ConcurrentDictionary to track found items. This requires synchronization to ensure thread safety while accessing and modifying the shared data structure. Using a ConcurrentDictionary allows for safe concurrent updates without explicit locking mechanisms, simplifying the synchronization process.

### 4. Are communications needed?

There are no significant data dependencies since each search task is independent. Threads can operate without waiting for each other to complete their tasks.

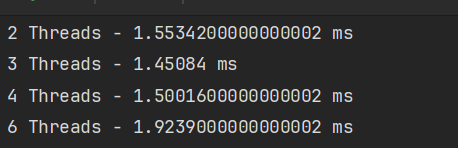
### 5. Are there synchronization needs?

Yes, synchronization is needed when updating the shared ConcurrentDictionary of found items. A lock can be used to ensure that updates to the dictionary and checks for search completion are atomic operations. This prevents race conditions and maintains data integrity during concurrent access.

### 6. Will load balancing be a concern?

Load balancing could be a concern if the distribution of item types is uneven. To optimize performance, the partitioning should ensure that each thread has a roughly equal workload.

## Task results and observations

.

The task results show varying average search times with different thread counts, with 2 threads taking 1.5534 ms, 3 threads achieving the best performance at 1.4508 ms, 4 threads at 1.5002 ms, and 6 threads increasing the time to 1.9239 ms. This suggests that while 3 threads provided the fastest search time, increasing the thread count beyond this may introduce overhead from context switching or load imbalance, indicating the need to find an optimal number of threads for efficient performance.