

**CIS6007 Parallel and Distributed Systems**

**Assignment A**

**Name: Alfiya Anjum**

**Student ID: st20317473**

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

**GitHub repository link:**

[**https://github.com/Alfiya-Anjum/Parallel-Systems\_Assignmnet**](https://github.com/Alfiya-Anjum/Parallel-Systems_Assignmnet)

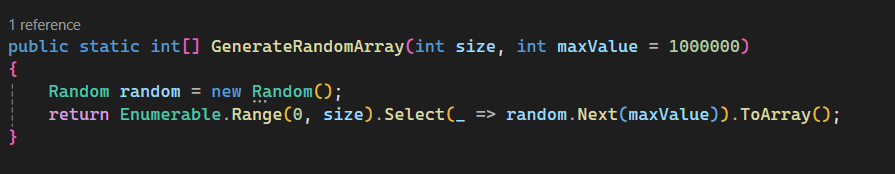
Task A

**Overview**

This program implements a parallel version of the Bubble Sort algorithm. The input array is divided into partitions, sorted independently using multiple threads, and merged into a single sorted array. Execution times are compared for different thread counts to evaluate the performance.

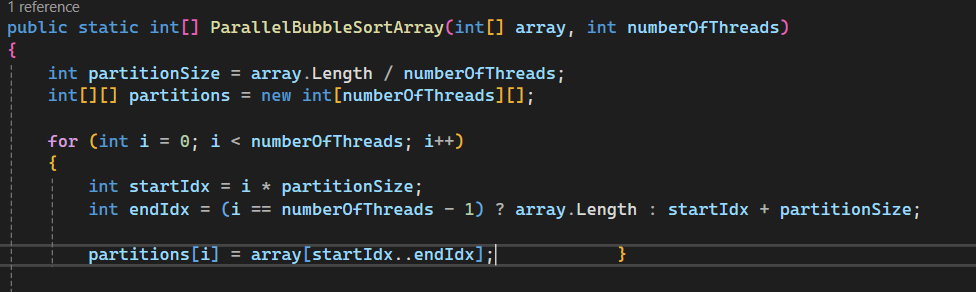
**Structure of the Program**

Random Array Generation:



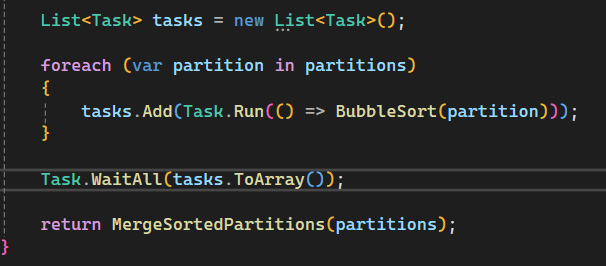
The GenerateRandomArray method creates an array of 100,000 random integers between 0 and 999,999. This array serves as the input for sorting.

Array Partitioning:



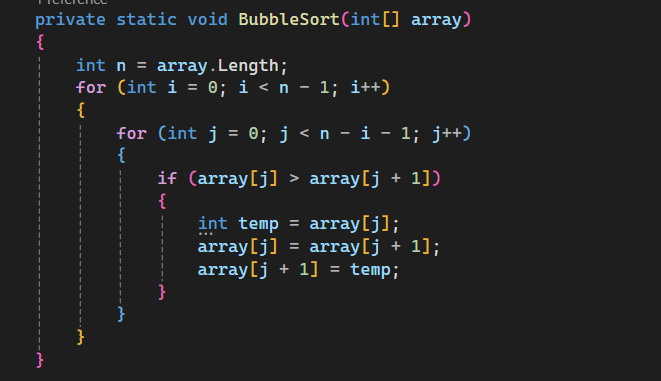
The input array is divided into equal-sized partitions based on the number of threads. Each partition represents an independent subset of the array to be sorted in parallel.

Parallel Sorting:



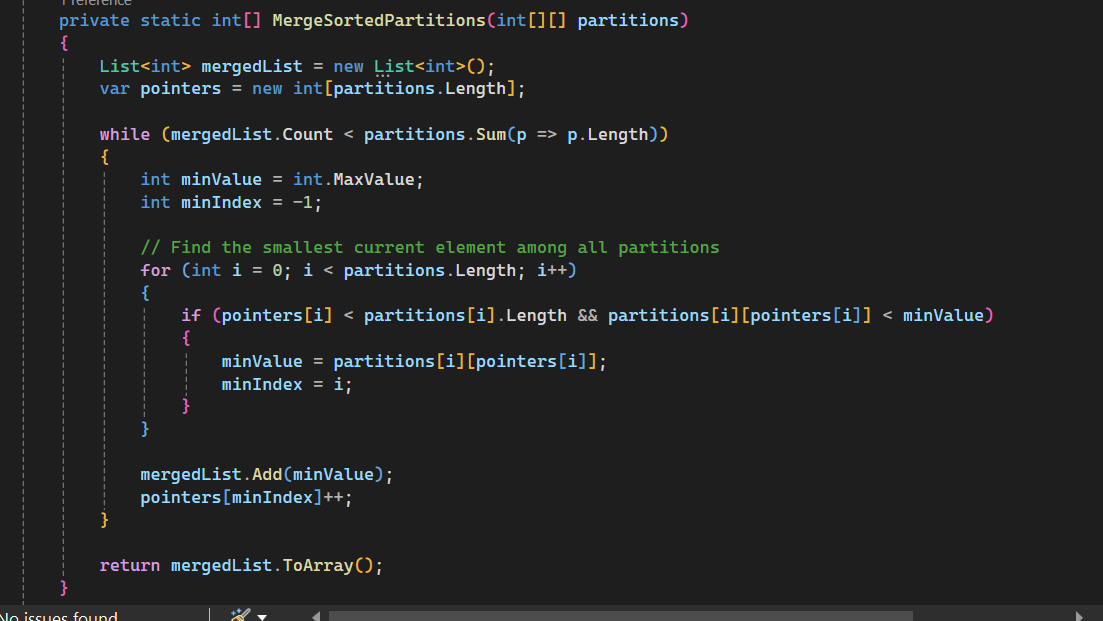
The Bubble Sort algorithm is applied to each partition independently. Using Task.Run, these sorting tasks are executed in parallel.

Bubble Sort:



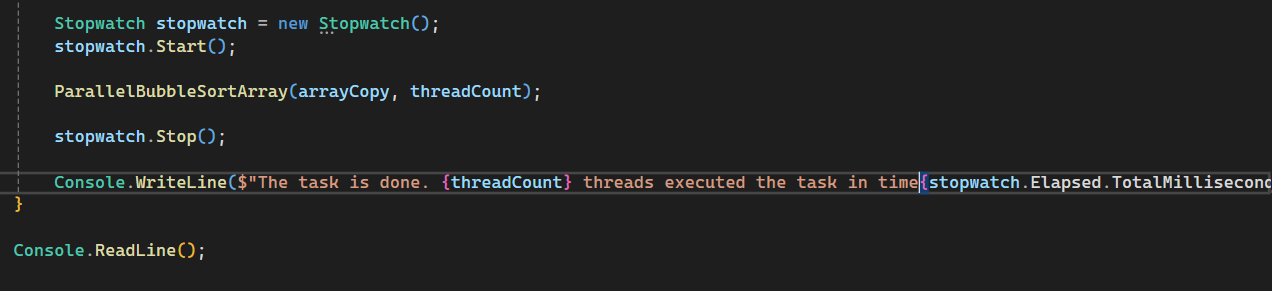
A simple Bubble Sort implementation sorts individual partitions.

Merging Sorted Partitions:



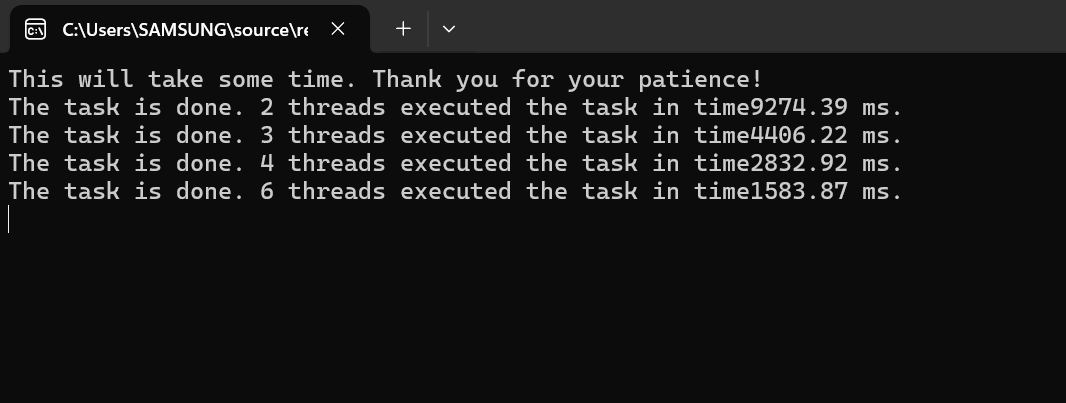
Once all partitions are sorted, they are merged into a single sorted array using a pointer-based approach to ensure global ordering.

**Performance Testing**:



The program measures the execution time for 2, 3, 4, and 6 threads. A stopwatch is used for precise timing.

**Sample Output**



The output demonstrates the execution time for sorting a large array of 100,000 elements using a parallel Bubble Sort algorithm. As the number of threads increases, the execution time decreases significantly, showcasing the effectiveness of parallelization. This result highlights the algorithm's scalability and efficiency when leveraging multiple threads for processing.

**Evaluation of the Task:**

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

Yes, this problem is well-suited for parallelization. The Bubble Sort algorithm can be applied to individual partitions of the array independently. By dividing the input array into smaller chunks and sorting them concurrently, the workload can be distributed across multiple threads. Since there is no dependency between the partitions during sorting, they can be processed in parallel without interference.

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

The problem is partitioned by splitting the input array into equal-sized chunks based on the number of threads. For example, if the array has 100,000 elements and 4 threads are used, each thread processes approximately 25,000 elements. The last partition accounts for any remaining elements if the array size is not evenly divisible by the number of threads. Each partition is then sorted independently.

**3. Are communications needed?**

No, communication between threads is not needed during the sorting phase. Each thread works independently on its assigned partition. The only interaction occurs after sorting when the partitions are merged into a single sorted array. However, this merging process does not require communication between threads, as it is performed sequentially.

**4. Are there any data dependencies?**

No, there are no data dependencies during the sorting phase. Each thread operates exclusively on its partition without relying on data from other partitions. This independence makes the sorting phase embarrassingly parallel.

**5. Are there synchronization needs?**

No, synchronization is not required during the sorting process, as each thread handles its partition independently. The merging step occurs after all threads complete their sorting tasks, so there is no overlap in operations that would require synchronization.

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

Load balancing is not a significant concern in this implementation. The partitioning logic ensures that all threads are assigned approximately equal numbers of elements to sort. However, due to the nature of the Bubble Sort algorithm, partitions with already sorted or nearly sorted data may complete faster than others. This discrepancy is minor in practice and does not significantly impact overall performance.

Task 2

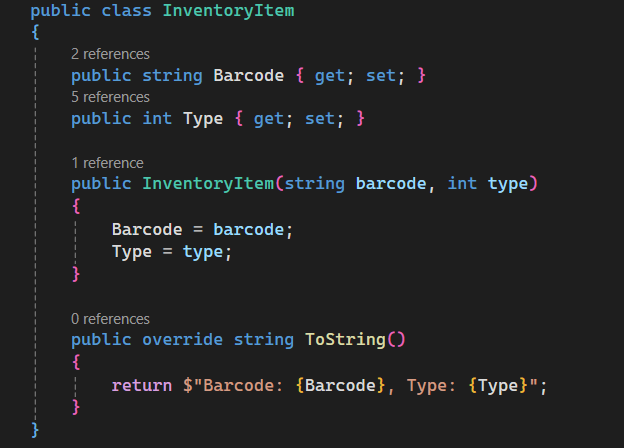
**Components of the Program**

**InventoryItem Class**

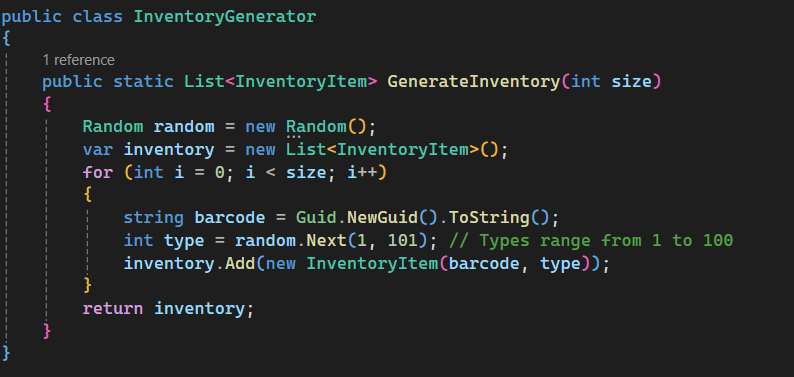
* **Purpose**: Represents the carpenter tools in the inventory.
* **Attributes**:

Barcode: A unique identifier for each tool (generated using Guid).

Type: An integer between 1 and 100 indicating the type of tool.

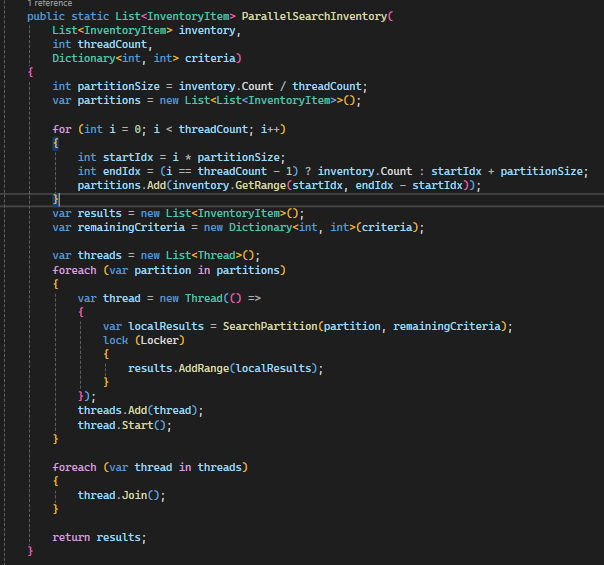


1. **Inventory Generation**
   * **Purpose**: Generates a large list of tools with random barcodes and types.
   * **Implementation**:
     + Uses a loop to populate the inventory with 100,000 random items.
     + Ensures unique barcodes and a wide variety of tool types.



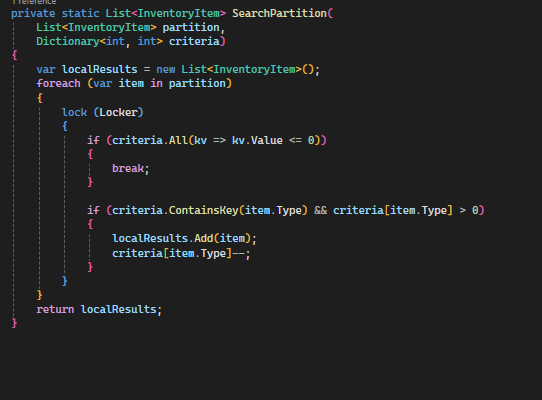
**Parallel Search Logic**

* + **Purpose**: Divides the inventory into chunks and performs a parallel search for specific items based on criteria.
  + **Key Features**:
    - Splits the inventory into equal-sized chunks for each thread.
    - Synchronizes access to shared resources using lock to avoid race conditions.



**Search Partition**

* + **Purpose**: Searches for matching items in a partition and updates shared resources.



**Main Program**

* + **Purpose**: Integrates the inventory generation and parallel search logic. Measures execution time and tests with varying thread counts.



**Evaluation of the Task**

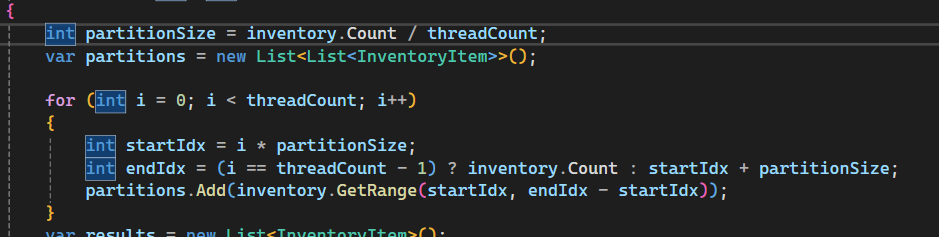
**1. Is this problem able to be parallelized?**

Yes, this problem is highly suitable for parallelization because the inventory can be divided into independent chunks. Each chunk can be processed by a separate thread, as the search for items in one chunk does not depend on the processing of other chunks.

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

The inventory is divided into equal-sized chunks based on the number of threads. For example:

* If the inventory contains 100,000 items and there are 4 threads, each thread will process 25,000 items.
* The last chunk may handle extra items if the total size is not evenly divisible by the number of threads.

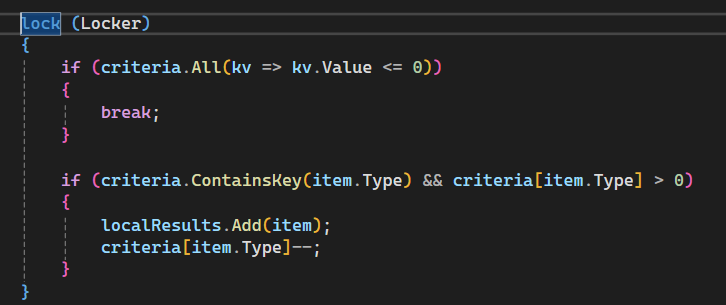
****

**3. Are communications needed?**

Minimal communication is required. Threads process their assigned chunks independently. However, there is shared access to the results list and criteria dictionary, which requires synchronization to avoid race conditions.

**4. Are there any data dependencies?**

Yes, there are dependencies on the shared criteria dictionary, which tracks the number of required items for each type. If one thread finds a matching item, it must update the dictionary so that other threads do not exceed the required count.

****

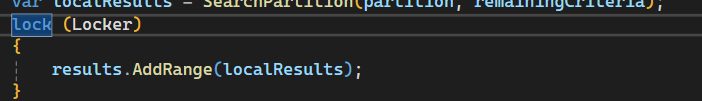
**5. Are there synchronization needs?**

Yes, synchronization is needed to:

1. Ensure threads safely update the shared results list and criteria dictionary.
2. Prevent race conditions when multiple threads try to access or modify the same resources.

**Solution**:

* Use a lock to synchronize access to shared resources.

****

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

Load balancing is not a major concern, as the inventory is divided into equal-sized chunks. However, some threads may finish earlier than others if their assigned chunks contain fewer matching items. The impact is minimal since all threads terminate as soon as all required items are found.

Solution output

The output showcases the performance of the parallel inventory search algorithm for an inventory of 100,000 items. The algorithm successfully retrieves all 53 required items (30 of type 1, 15 of type 7, and 8 of type 10) with different thread counts (2, 3, 4, and 6). The execution times vary slightly across thread counts, indicating the impact of thread management and system overhead on parallel performance. Despite the variations, the program demonstrates efficient and accurate retrieval of the specified items.