Assignment #1

Tags | Assignment / Exam | Parallel & Dist. Systems

Information

Name: Vladimir W. Schmadlak

Student ID: st20283948

Programme: BSc (Hons) Software Engineering

Github: https://github.com/VladimorCodebreaker/Parallel-Systems-Assignment.git

Task 1

Structure of the Program

- **ParallelBubbleSorter.cs**: Contains the implementation of the parallel bubble sort algorithm.
  It includes methods for generating a random array and performing the parallel bubble sort.
- **SortPerformanceAnalyzer.cs**: Analyzes the performance of the parallel bubble sort.
  It measures the execution time for sorting the array with different thread counts.

Evaluation of Task 1

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

Yes, the bubble sort algorithm was parallelized by dividing the array into sub-arrays and sorting them in parallel.

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

The array is divided into equal-sized sub-arrays based on the number of threads. Each sub-array is sorted independently.

3. **Are communications needed?**

Minimal communication is required, primarily for merging the sorted sub-arrays.

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

Each sub-array is independent in the sorting phase, reducing data dependencies. However, dependencies arise during the merging phase.

5. **Are there synchronization needs?**

Synchronization is needed to ensure that all sub-arrays are sorted before they are merged.

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

Load balancing is handled by equally dividing the array among the threads. Though, the efficiency of load balancing depends on the uniformity of the data distribution.

Test Results

```
Starting Task 1: Parallel Bubble Sort

Threads: 2, Time: 6658.0987 ms
Threads: 3, Time: 2945.831 ms
Threads: 4, Time: 1631.1836 ms
Threads: 6, Time: 709.1707 ms
```

Observations

- The test results demonstrate effective parallelization, with a clear trend of decreasing execution time as the number of threads increases. The most significant performance improvement is observed when using 6 threads, indicating that the algorithm scales well.

Task 2

Structure of the Program

- **Tool.cs**: Defines the `Tool` class with properties for `Type`, `Barcode`, and `Description`.
- **InventoryGenerator.cs**: Generates a list of `Tool` objects representing an inventory. Each tool has a type, a unique barcode, and a description.

- **ParallelInventorySearcher.cs**: This file contains the implementation of the parallel search algorithm. It employs the `Parallel.ForEach` method, which allows the algorithm to run across multiple threads simultaneously. The degree of parallelism, i.e., the number of threads used in the search process, is adjustable and determined by the input provided.
- **SearchPerformanceAnalyzer.cs**: Measures and analyzes the performance of the parallel search algorithm with different thread counts.
- **Program.cs**: Orchestrates the execution, including generating inventory, defining search targets, executing the performance analysis, and handling user interaction.

Evaluation of Task 2

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

Yes, the problem is well-suited for parallelization. The search for different tool types in a large inventory can be efficiently performed in parallel, as each search operation is independent.

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

The inventory list is partitioned across multiple threads. Each thread processes a portion of the inventory to search for the required types of tools.

3. **Are communications needed?**

Minimal communication is needed. While threads operate independently, they update a shared `ConcurrentDictionary` that tracks the found items, requiring synchronization for thread safety.

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

No significant data dependencies exist between the threads. Each thread can independently search for tools without relying on the results of other threads.

5. **Are there synchronization needs?**

Yes, synchronization is needed when updating the shared `ConcurrentDictionary` of found items. A lock (`_lockObject`) is used to ensure that updates to the dictionary and checks for search completion are atomic operations.

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

Load balancing might be a concern, depending on the distribution of tool types in the inventory. If certain types are clustered together, some threads might have more work than others. However, `Parallel.ForEach` handles workload distribution, which mitigates this concern to some extent.

Test Results

The performance of the parallel inventory search algorithm was tested with different thread counts (2, 3, 4, and 6). The average time taken for each thread count was measured, and the correctness of the search results was verified.

```
Starting Task 2: Parallel Inventory Search

Threads: 2, Average Time taken: 1.95916 ms
Type 1, Found: 30, Target: 30
Type 7, Found: 15, Target: 15
Type 10, Found: 8, Target: 8

Threads: 3, Average Time taken: 2.6059599999999996 ms
Type 1, Found: 30, Target: 30
Type 7, Found: 15, Target: 15
Type 10, Found: 8, Target: 8

Threads: 4, Average Time taken: 3.0457 ms
Type 1, Found: 30, Target: 30
Type 7, Found: 15, Target: 15
Type 10, Found: 8, Target: 8

Threads: 6, Average Time taken: 3.30972 ms
Type 1, Found: 30, Target: 30
Type 7, Found: 15, Target: 15
Type 10, Found: 8, Target: 8
```

Observations

- The algorithm correctly identified the specified number of items for each tool type in every test, validating its accuracy.
- The average execution time increased with the number of threads. This could be attributed to the overhead of managing more threads and potential contention for shared resources, notably the lock used in the algorithm.