

**CIS6007 Parallel and Distributed Systems**

**Assignment A**

**By**

**Name: Filip Svetlomirov Petkov**

**Student ID: ST20283945**

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

**Github repository link:**

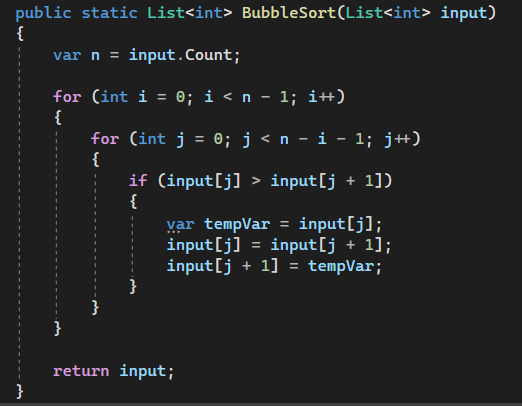
<https://github.com/Hellsfan/Parallel_Distributed_Systems_Assignment_1.git>

**Task 1:** Implement a parallel version of the bubble sort sorting algorithm

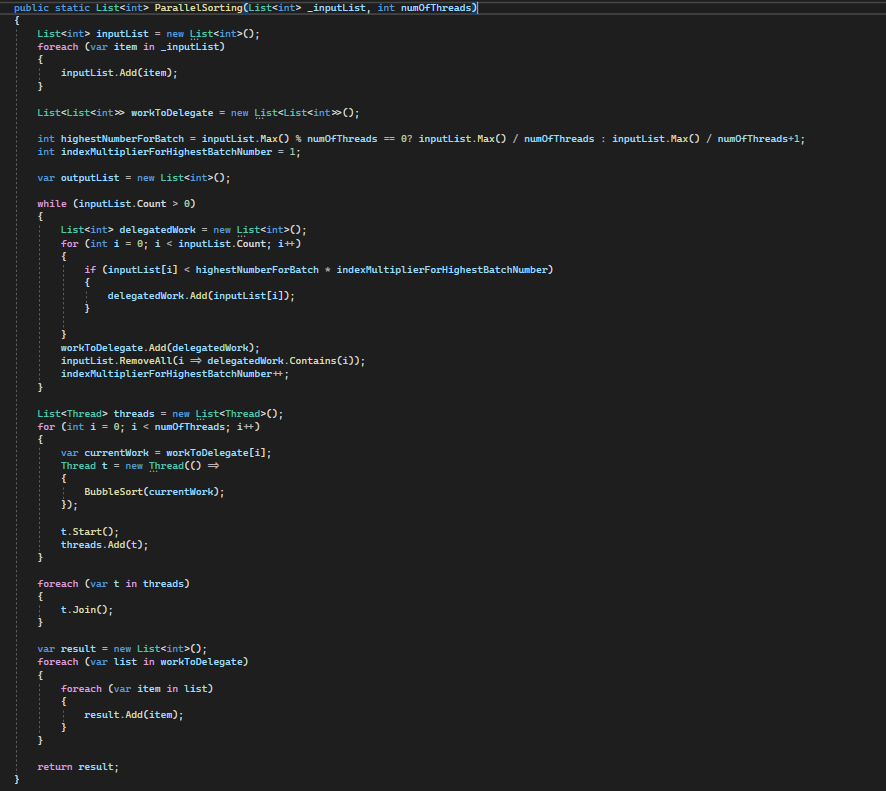
Explanation of the task: Bubble sorting is a type of sorting algorithm of an array of items. It takes two items, compares them by a specific criteria and if such criteria is fulfilled swaps their positions. The algorithm is finished when no swaps happen from the first element of the array to the last. The algorithm can be easily implemented with multiple threads as well, and no issues occur throughout the process.

**-Structure of the code:**

I implemented a very straightforward and standard version of a bubble sort function which takes a list of integers and returns it sorted.

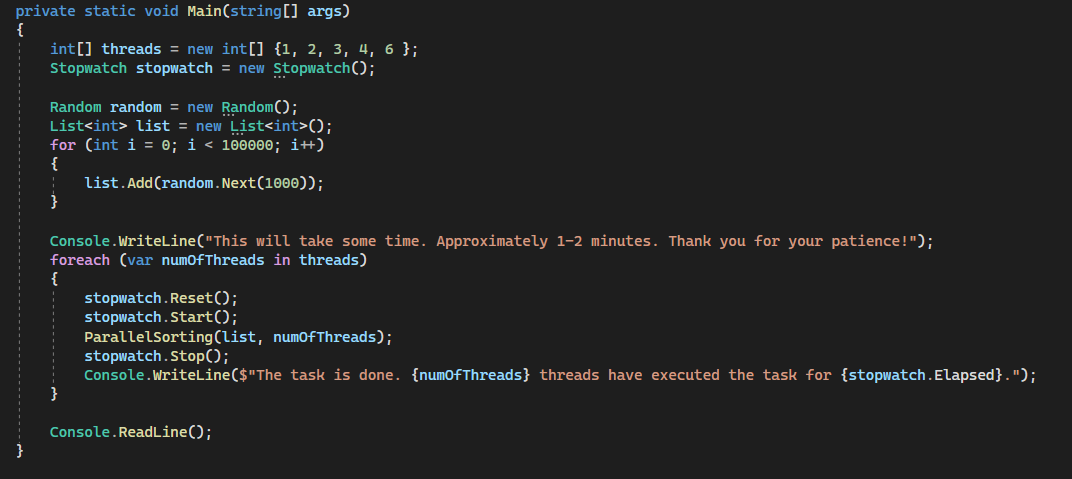


For the Parallel implementation of the algorithm I made the following code:



We take the list we want to sort and the amount of threads we want to use. Then we split the initial list to smaller batches, which will be delegated to the same amount of threads. The way the splitting works is by calculating a maximal item for each batch. For example in the case of 100 000 integers with 5 batches, the first batch would get all numbers from 1 to 200 000. The second batch would get all numbers from 200 001 until 400 000 and so on. That way we ensure that when all the batches are combined into one list at the end of the algorithm, they will automatically align without the need of sorting them again. Afterwards we create threads and assign each batch to them and run them. In the end we combine all the batches into a big list and return it.

For the testing of the code I wrote the following:



I used a stopwatch object to track time and a random object to generate the list. Then we just call the function for the amount of threads we want to test and wait for the results.

**-Evaluation of the task according to the following criteria**

-Is the problem able to be parallelized?

Yes. The Bubble sort algorithm is pretty straightforward by itself. So if we just split the work into batches and give it to each thread no issues happen. The threads work independently of each other without interfering into eachothers resources.

-How would the problem be partitioned?

As already mentioned, one of the ways is through calculating a maximal number for each batch depending on the number of threads. We take the array of integers and divide it by the number of threads. In case the splitting is a decimal number, we round it to the ceiling, which means some batches would have one more number to sort. Afterwards each batch receives all the items that are lower than the splitting point and all those items get removed from the main array. This goes on, until all the numbers in the main array are put into batches.

-Are communications needed?

No. In my implementation the most dangerous part is how to split the work into batches. Afterwards, if everything is split correctly, the threads have no need for synchronization and they can work completely independent of each other. Then the batches are combined into one big list and if the splitting was correct to begin with, all the items should be automatically sorted from start to finish.

-Are there any data dependencies?

No. The threads work on their own individual batch, without the need of interfering with each other.

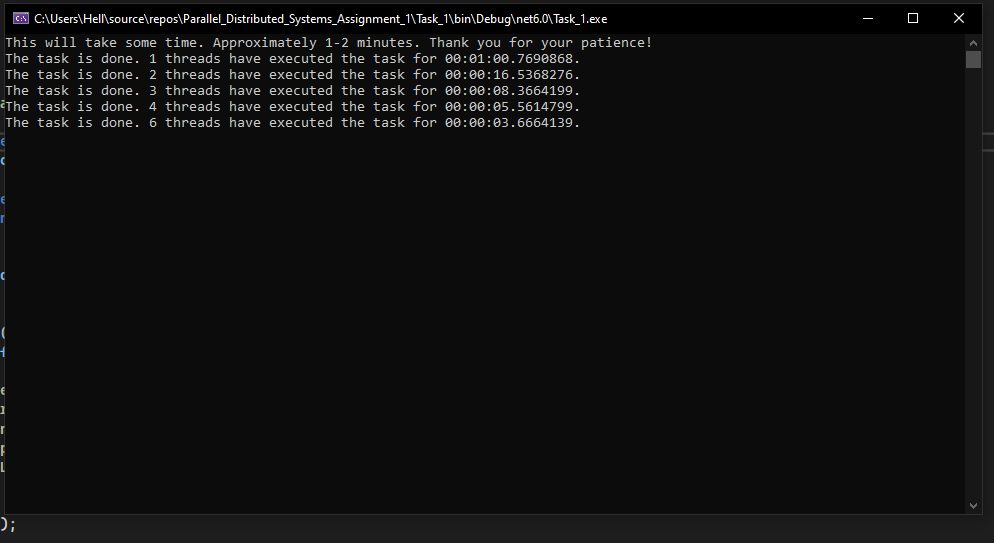
-Are there synchronization needs?

No. In my implementation no such things as locks or communication between threads were needed, therefore the threads can work completely asynchronous.

-Will load balancing be a concern?

Depends. The calculation of partitions should eliminate this problem, because it splits batches into equal parts and all threads get an equal amount of numbers to sort. Though it is possible due to randomization, that one thread gets a “better” randomized list,, which would mean less iterations of the bubble sort. But overall, the algorithm is simple enough to conclude the following: the more threads we use, the fast the algorithm is.

**-Test results**



First of all we can safely conclude that the algorithm benefits immensely from multi-threading. Initially with 1 thread it takes approximately 1 minute to finish. But with 2 threads we see almost 4 times the speed comparing it with the initial 1 thread. Afterwards the more threads we add the faster it gets, although time differences would not see such substantial growth. In conclusion the Bubble sort algorithm can be completely parallelized and the outcome of it increasing processing speed gain.

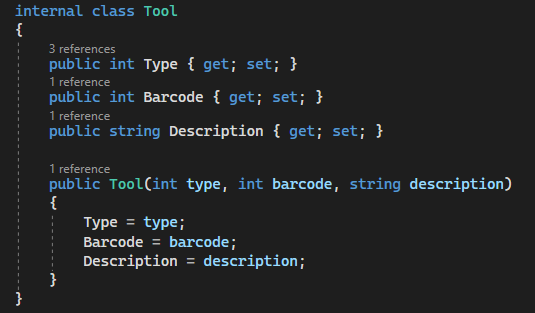
**Task 2:** Implement a parallel version of a search algorithm of a large list of items

Explanation of the task:

The task is to develop an algorithm that searches through a list of objects and returns a specific amount of tools of a specific type based on criteria. Afterwards a parallel implementation should be developed, which would use threads to make the searching possibly faster or slower.

**Structure of the code:**

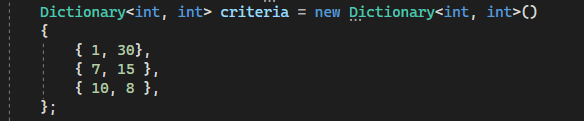
First I created a simple tool object which would be used for the list of items:



Then I created an Inventory class which has a list of Tools. When the constructor of the class inventory is called it creates a randomized list of 100 000 tools to be used for the algorithm:



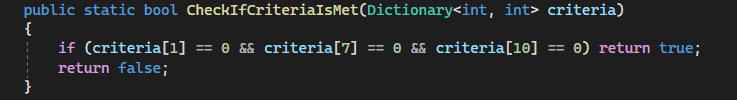
These were the necessary objects that will be used in the code. Now the main part of the code begins with the following implementation of the criteria of what kind of tools we need to get from the list:



This is needed to keep track of how many items of specific type we have retrieved from the list.

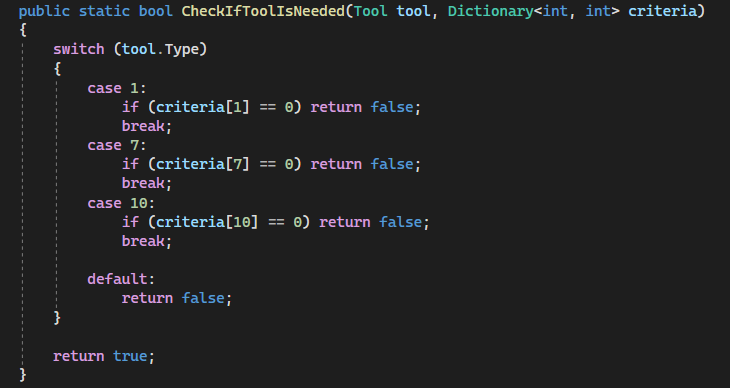
Afterwards I have created two functions which will check for specific events that are important for the algorithm

1.Check if the criteria is met:



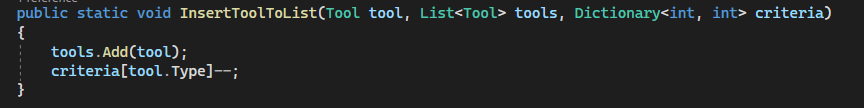
This is used to tell us that we have gathered all the necessary tools from the list and it is time to stop the algorithm, because no more tools are needed.

2. Check if this specific tool is needed:



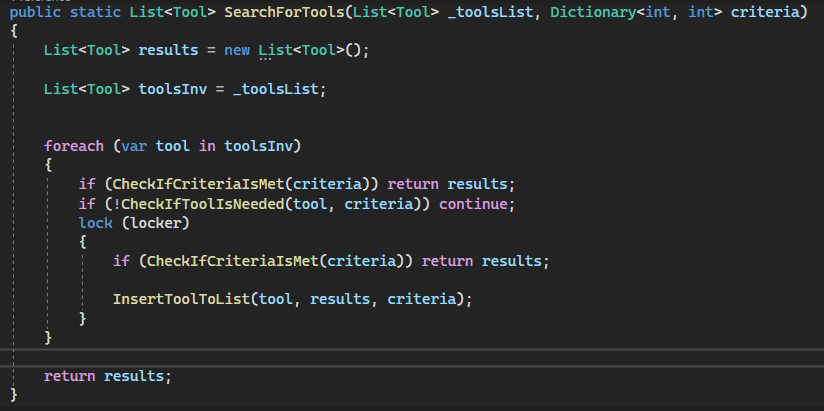
This is used to check if this specific tool is needed every time a tool is retrieved from the list. It will check inside the criteria dictionary with the specific key. Then if the value is higher than zero, it means that the tool is needed and we return true. Otherwise if the value is 0 or the key does not exist ( the default case ) we return false, as this specific tool is not needed.

Then we have the simple, but most dangerous function of all: adding a tool to the list and editing the criteria.



This is important and dangerous, because threads will access the same dictionary to keep track of the criteria, as well as editing it. This function is also the one that requires locking logic, because one thread at a time should be able to edit the criteria.

Now here is the simple implementation of the Searching algorithm:



The algorithm is adapted so it can work with threads as well. We create 2 lists. One for results and one to copy the input list to. Afterwards we check each item in the inputList.

First we check if the criteria is met:

If true —> stop the algorithm and return results, because we are done

If false —> we continue.

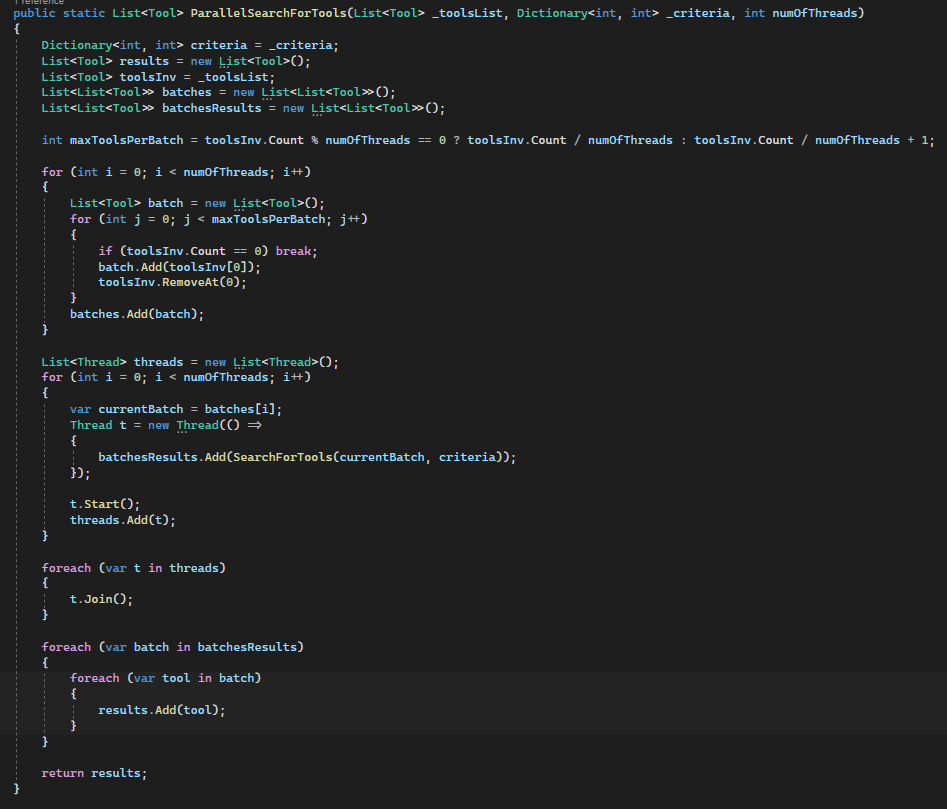
Then we check if this specific tool is needed:

If false —> continue with next item and skip following code

If true —> go inside lock statement

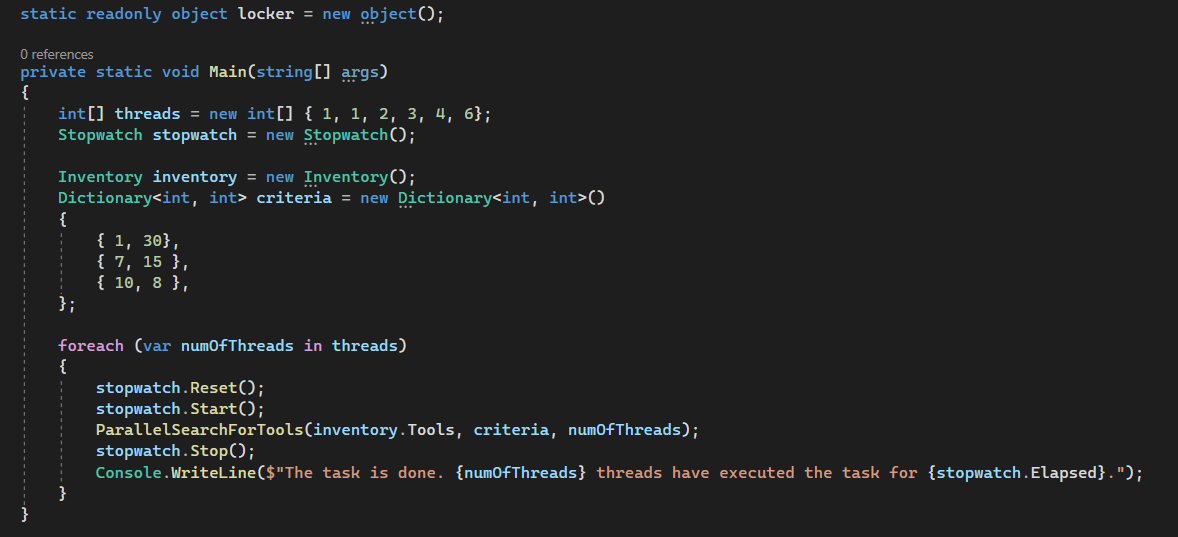
So once a thread is inside the lock statement we once again check the criteria is met. This is done, because 1 thread can enter inside the lock statement, thus allowing us to safely check one last time if the criteria is met or not. Afterwards the insertion happens and we continue with the next tool in the list. In the end we return the results list.

Finally we reach the multi-threading part. The code is the following:



I used the same logic from task 1 to split the workload into batches and then give each thread a batch to check into for the required tools. Partition happens through dividing the amount of tools by the number of threads. Then we initialize and start all threads, where each thread has its own batch to work with. The locking statement inside the Searching function guards us from possible race conditions, where multiple threads would want to insert tools and edit the criteria at once. Once all the threads are done we combine the results and return them.

To test everything the code is the following:



Again we use a stopwatch to track time and a locker object for the locking logic. We provide one criteria dictionary to all threads which they use to synchronize when to stop.

**- Evaluation of the task based on the following criteria:**

-Is this problem able to be parallelized?

Yes. The problem can be parallelized if we use the same batch logic. Each thread receives their own workload and all we need is some simple synchronization in order to control them via the criteria dictionary.

-How would the problem be partitioned?

The problem would be partitioned by splitting the list of tools by the number of threads, as already shown in the code. That way each thread receives an equal amount of tools to search through.

-Are communications needed?

Yes. A locking logic. Otherwise the criteria dictionary would be overwritten at all items by multiple threads at once and complete chaos would happen.

-Are there any data dependencies?

Yes. The Criteria dictionary, because all the syncing logic revolves around it, it is of utmost importance to manage it properly.

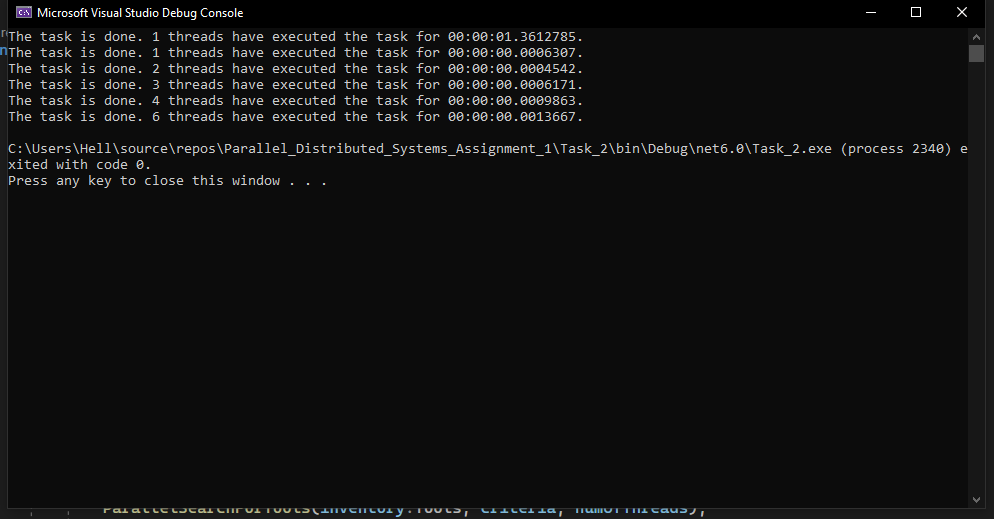
-Are there synchronization needs?

Yes. Whenever an item is needed and is going to be added, it should happen synchronously. That way it happens one item at a time and we can control how many items are added at all times. And of course stop the addition of items when needed.

-Will load balancing be a concern?

Possibly yes. Due to randomization, most of the tools can be partitioned in such a way that one batch can contain all the tools and other batches can contain zero tools. In such a case one thread has all the workload, while the rest will do nothing. Though there are possible ways to optimize that and resolve such problems.

**Test results**



Due to the speed of the algorithm, we have one additional execution with 1 thread to warm up the memory. In this implementation of the algorithm with multi-threading we get slower execution speeds the more threads we add. With 1 thread the algorithm is the fastest. With 2 threads, sometimes the algorithm is faster than the one with one thread, but most of the time it is slower. Afterwards the more threads we add, the slower it gets. This is supposedly due to the locking logic. The searching happens so fast, that forcing the threads to wait for each other when adding a tool happens, just makes them form a queue and slows down everything. In conclusion a parallel implementation here doesn’t seem effective, due to the need of synchronization between threads which makes unnecessary wait time between them.