Assignment 2

Concurrency and multithreading 2016

Dimitri Diomaiuta – 2553339
Hugo Brandao - 2553124

Introduction:

This assignment is about Fine grained locking. Fine grained locking can be seen as an improvement over coarse grained locking. This implementation, in fact, does not lock the whole data structure to make it thread safe, it locks instead just the parts of the structure needed by each thread to perform update operations (in this case adding or removing elements). This allow multiple threads to access the same data structure at the same time and update it since only parts of it will be locked by each thread. There is still the probability that some threads will have to wait to acquire a lock on a an object but overall the thread concurrency allowed by this implementation is much higher than the previous one. The two data structures used are the same as in the previous assignment: singly linked list and binary search tree.

Hypotheses:

We now formulate three hypothesis on how the Fine grained implemented versions of the two data structures will behave in comparison to the previous implementations (coarse grained ones), each other (fine grained list vs fine grained tree) and having more or less parallelization allowed (the work time parameter):

- **H1:** Increasing the work time parameter does not result always in slower execution. Running the same data structure with the same parameters (number of threads, number of items) but an increased work time parameter may result in a faster execution since avoids too many context switches from one thread to another.
- **H2:** Knowing that the fine grained data structure allow a much higher thread concurrency we can hypothesize that with the same number of threads and number of items the same data structures will perform faster in their fine grained implementation than in the coarse one. The work time parameter will be set differently for the different data structure accordingly to the value that provides a fastest execution. The number of threads must be bigger than one to see the concurrency gain.
- **H3:** Knowing that the tree data structure provides on average faster access than the list one we can hypothesize that with the same parameters (number of threads, number of items) the fine grained tree will perform faster than the fine grained list. The work time parameter will be set differently accordingly to the value that provides a fastest execution.

Evaluation:

Support or disprove of the hypotheses statements was done by testing the implemented data structures, with different parameters, and analyzing the gathered data. The environment used for running the implemented data structures was the DAS4 cluster. We used, as in the previous assignment, the averageTime.sh script (appendix A) to have an average of 2 different runs of the same program in order to have more accurate results (we changed to 2 since number of elements used this time was much bigger and hence executions slower).

H1 evaluation:

The first hypothesis was evaluated by running the fine grained implemented data structure with the same parameters but an increased **work time** parameter each time (we initially increase it to 5 and if there are speed gains we continue increasing it otherwise we decrease it). In this evaluation we tested both the list and the tree in their fine grained implementation. The **number of items**, for each data structure, was chosen high enough to cause the program to run at least for more than one second. This in order to make it possible to see speed ups after changing the work time parameter. The **number of threads** was chosen in order to start from 2 and increase in different steps to reach 1% of the number of elements (that means that each threads deals with 100 elements) in order to see how the work time has different effects on different number of threads. The data gathered from testing can be seen in table 2.1

Structure	Implementation	No of threads	No of items	Work time	Average time results (20 runs) in ms
Tree	Fine	2	1.000.000	0	3461
Tree	Fine	2	1.000.000	5	8973
Tree	Fine	2	1.000.000	1	4894
Tree	Fine	8	1.000.000	0	9233
Tree	Fine	8	1.000.000	5	8136
Tree	Fine	8	1.000.000	10	5032
Tree	Fine	8	1.000.000	15	5014
Tree	Fine	8	1.000.000	20	6247
Tree	Fine	10	1.000.000	0	9049
Tree	Fine	10	1.000.000	5	8480
Tree	Fine	10	1.000.000	15	6037
Tree	Fine	10	1.000.000	20	5599
Tree	Fine	10	1.000.000	30	7209
Tree	Fine	100	1.000.000	0	5238
Tree	Fine	100	1.000.000	5	9402
Tree	Fine	100	1.000.000	1	7948
Tree	Fine	1000	1.000.000	0	7630
Tree	Fine	1000	1.000.000	5	9134
Tree	Fine	1000	1.000.000	1	8482
Tree	Fine	10.000	1.000.000	0	15789
Tree	Fine	10.000	1.000.000	1	15970

List	Fine	2	20.000	0	6288
List	Fine	2	20.000	5	6730
List	Fine	2	20.000	1	6830
List	Fine	8	20.000	0	1828
List	Fine	8	20.000	5	2616
List	Fine	8	20.000	1	2153
List	Fine	20	20.000	0	1694
List	Fine	20	20.000	5	1729
List	Fine	20	20.000	1	1697
List	Fine	100	20.000	0	1490
List	Fine	100	20.000	5	1608
List	Fine	100	20.000	1	1576
List	Fine	200	20.000	0	2027
List	Fine	200	20.000	5	1757
List	Fine	200	20.000	10	1777

Table 2.1

As it can be seen from table 2.1 the work time parameter behavior depends on the number of threads used and on the type of data structure. We can see that it never affects the list execution to have an high speed gain, instead it affects the tree execution to have an high speed gain on small number of threads. When used 8 or 10 threads the tree structure can execute almost 2 times faster when the work time is increased. The obtained data confirms our hypotheses that increasing the work time parameter does not always slow down the program but can increase its execution speed. An increased work time parameter result in a speed gain when a small number of threads is used since it makes the program having less context switches. When the number of threads is too small or too big (in our case < 8 and > 10) the work time context switch gain is lower than the overhead it adds to each thread, hence it result in a slower execution. The reason why the speed gain is almost nothing in the list is that, differently from the tree, threads must visit the same path each time (list does not have many levels as a binary search tree) so the more a thread works the higher the probability of holding a lock on an element needed by another thread. In this case the context switch gain is lower than the busy waiting cause by threads increased work time.

H2 evaluation:

The second hypothesis was evaluated by testing each data structure fine grained implementation with its coarse one. The **number of items** for the tests was chosen high enough to make the data structures run for more than one second in order to be able to record speed ups. The **number of threads** was chosen to start from 2 and increase in different steps to reach 1% of the number of elements (that means that each threads deals with 100 elements) in order to make comparison of the different implementations using different number of threads. The **work time** parameter was chosen accordingly to the one making the data structure to run faster. When comparing a data structure fine grained implementation with a coarse one the parameters are all the same except from the work time that can be different. The data gathered can be seen in table 2.2.

Structure	Implementation	No of threads	No of items	Work time	Average time results (20 runs) in ms
Tree	Coarse	2	1.000.000	0	6294
Tree	Fine	2	1.000.000	0	3461
Tree	Coarse	8	1.000.000	0	5882
Tree	Fine	8	1.000.000	15	5014
Tree	Coarse	10	1.000.000	0	5689
Tree	Fine	10	1.000.000	20	5599
Tree	Coarse	100	1.000.000	0	5538
Tree	Fine	100	1.000.000	0	5238
Tree	Coarse	1000	1.000.000	0	6096
Tree	Fine	1000	1.000.000	0	7630
Tree	Coarse	10.000	1.000.000	0	10.804
Tree	Fine	10.000	1.000.000	0	15.789
List	Coarse	2	20.000	0	3453
List	Fine	2	20.000	0	6288
List	Coarse	8	20.000	0	3208
List	Fine	8	20.000	0	1828
List	Coarse	20	20.000	0	3678
List	Fine	20	20.000	0	1694
List	Coarse	100	20.000	0	2374
List	Fine	100	20.000	0	1490
List	Coarse	200	20.000	0	2061
List	Fine	200	20.000	10	1777
List	Coarse	1000	20.000	0	2682
List	Fine	1000	20.000	0	5316
List	Coarse	10.000	20.000	0	9849
List	Fine	10.000	20.000	0	15.840

Table 2.2

We decided to add an high number of threads also on list, even though it is bigger than 1% of the number of items, in order to make our data more relevant and useful to prove/disprove our hypothesis. As it can be seen from table 2.2 the data gather by the tests do not confirm our second hypothesis completely. It can be seen, in fact, that when the same data structures are executed with an high number of threads (1000 or 10.000 in our tests) the coarse grained implementation performs better than the fine grained one. This is because of **cache misses**. Cache misses occur every time a new thread operates for the first time since its local cache is empty. This initial cache miss is shared by both the coarse grained implementation and the fine one and causes an equal overhead. What causes the fine grained implementation to have more overhead than the coarse one are subsequent cache misses. The fine grained implementation, in fact, do not work on the whole data structure but only on parts of it. This causes more cache misses than the coarse implementation where each thread works on the whole structure. The overhead caused by this behavior makes the fine grained data structures run slower than the coarse one on an high number of threads (in our tests more than 1000 threads). We can finally state that the data gathered during the tests does not confirm our hypothesis hence when run with the same parameters the fine grained implementation is not always faster than the coarse one.

H3 evaluation:

The third hypothesis was evaluated by testing the fine grained tree against the fine grained list. The two data structures where tested with the same parameters except for the work time one that was chosen accordingly to the one making the data structure to run faster. The **number of items**, as in the other evaluations, was chosen high enough to make at least one of the two data structure run for a few seconds in order to be able to see speed ups. The **number of threads** was chosen to start from 2 and increase in different steps to reach 5% of the number of elements (that means that each threads deals with 20 elements) in order to make comparison of the different implementations using different number of threads. The **work time** parameter was chosen accordingly to the one making the data structure to run faster, which in this case was 0 for all of the different executions. The results can be seen in chart 2.3.

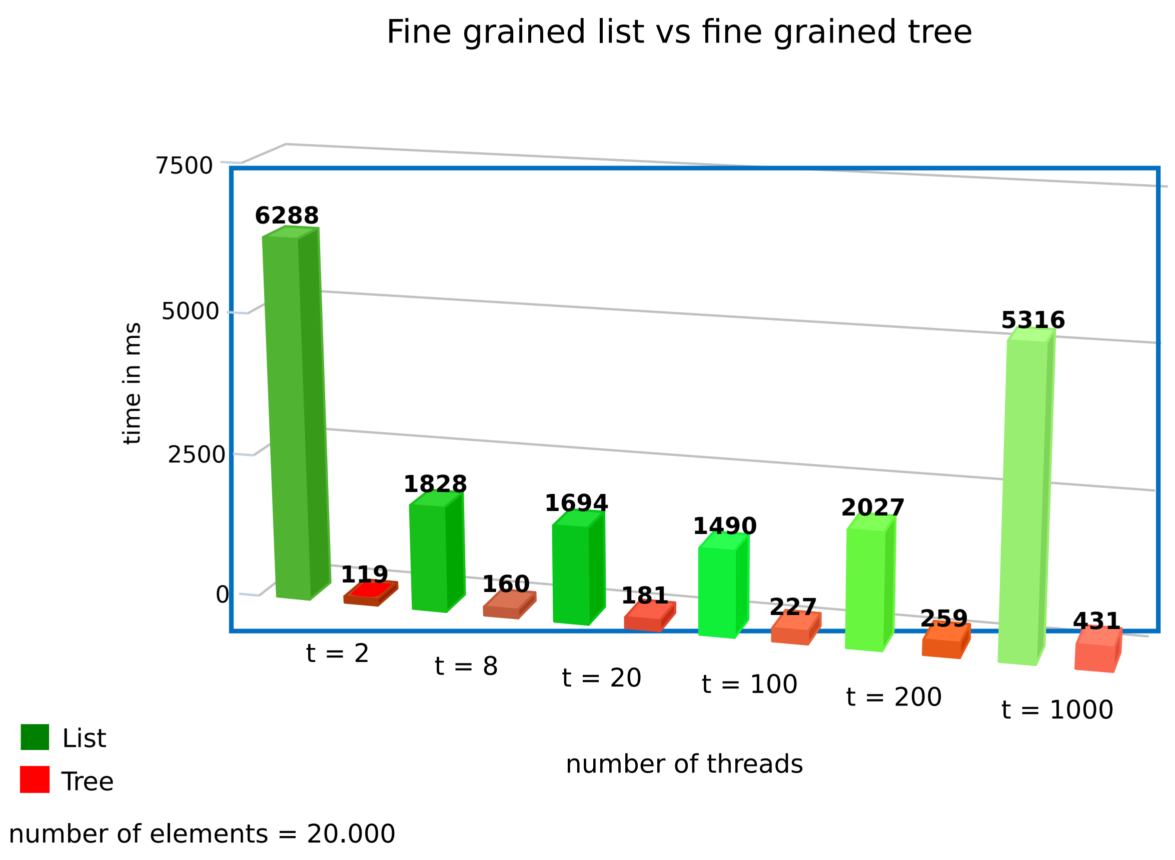

Chart 2.3

As it can be seen from chart 2.3 the fine grained tree outperforms the fine grained list. It runs always from 5 to 10 times faster than the list. This proves our hypothesis of having the fine grained tree implementation executing faster than the fine grained list when executed with the same number of parameters. Each thread is, in fact, accessing the tree structure faster than the threads accessing the list one. This makes the fine grained tree performing faster than the fine grained list.

Appendix A:

The following script runs the input test data structure program 2 times and outputs the average of each execution time.

The script is run as follows: `./averageTime.sh "./bin/test_data_structures p1 p2 p3 p4"`

This is the script code:

```
#!/bin/bash

command="$1"

for i in $(seq 1 2)

do

    y=$($command | grep time | awk '{print $2}')

    x=$((x+y))

done

average=$(echo "$x/2" | bc -l)

echo "for the command $command the average is: $average"
```