Assignment 1

Concurrency and multithreading 2016

Dimitri Diomaiuta – 2553339
Hugo Brandao - 2553124

Hypotheses:

This assignment is about Coarse grained locking. Coarse grained locking is about translating a sequential implementation to a parallel one by blocking the whole data structure when a thread wants to access it. The data structures used in this assignment are singly linked list and binary search tree. Knowing the time complexity of these two data structure we can make the first hypothesis on how they will behave after being translated to coarse grained locking:

- **H1:** Knowing the time complexity of the data structures (see table 1.1) we can hypothesize that running the coarse grained binary search tree list and coarse grained singly linked list with the same parameters will result, on average, faster. This because the data structure is accessed by one thread at a time so no gain by multiple threads is achieved in data structure operation. Having faster data structure operation makes the binary search tree data structure faster with the same parameters as the singly linked list.

Data structure	Time complexity (average)				Time complexity (worst)			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Singly linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(1)	O(1)	O(n)
Binary search tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)

Table 1.1

- **H2:** Running the program from one thread to n threads will have a speed gain and will result in a faster execution since the items used to populate the data structure will be generated in a parallel way. Furthermore also the reads of the elements to remove from the data structure will be done in a parallel way. We can state that running the program with the same parameters but an increased number of threads will result in a faster execution.

Evaluation:

The evaluation phase is about gathering data to be analyzed in order to support or disprove the hypotheses statements. The data gathered was the execution time under different parameters and different data structure used. The environment used for running the implemented data structures was the DAS4 cluster.

When a program is run its execution time depends on multiple factors. This can result in slightly different timing result. In order to improve time results accuracy we decided to average these results by running the programs inside a bash script that would run each program for 20 times and output

the average of the execution time. The script, named averageTime.sh, can be found in appendix A.

H1 evaluation:

The first hypothesis was evaluate by running the two different data structures (coarse grained tree and coarse grained list) with the same number of parameters different times and analyze the execution time to confirm or not our hypotheses. We first run the two different data structures with one thread to see that the binary search tree was actually faster than the singly linked list. After that we run the programs with an increasing number of threads to analyze if the tree data structure was performing faster than the list one when both run with the same increased number of threads. The data gathered can be seen in table 2.1.

Structure	No of threads	No of items	Work time	Average time results (20 runs) in ms
List	1	200	1	7.450
Tree	1	200	1	5.150
List	1	400	1	16.750
Tree	1	400	1	9.050
List	2	200	20	11.050
Tree	2	200	20	8.700
List	2	400	1	20.150
Tree	2	400	1	9.250
List	10	400	1	23.250
Tree	10	400	1	14.350
List	20	400	1	23.550
Tree	20	400	1	16.650
List	400	400	1	84.400
Tree	400	400	1	60.000
List	50	10000	1	890.350
Tree	50	10000	1	157.350

Table 2.1

As it can be seen from table 2.1 the tree data structure runs always faster than the list one when having both the same parameters. This because, on average, the tree algorithm grows following a logarithmic function, on the other hand the list algorithm grows following a linear function. The more the elements used by the two data structure the biggest the gap in performance speed because of the different function growth.

H2 evaluation:

The second hypothesis was evaluated by running the two different programs with the same parameters but with a constantly increasing number of threads till the point where the number of threads and number of items parameters were equal. The evaluation, as in the previous hypothesis evaluation, were performed by running the programs through the averageTime.sh script (see Appendix A). The work time was this time increased for all the tests because the effort of this one

was not showing that one data structure performs faster than the other but that adding more threads to the same program with the same parameters was making the program running faster. The data gathered can be seen in table 2.2.

Structure	No of threads	No of items	Work time	Average time results (20 runs) in ms
List	1	400	40	45.050
List	2	400	40	30.100
List	8	400	40	26.100
List	10	400	40	27.050
List	20	400	40	28.500
List	40	400	40	29.250
List	100	400	40	33.500
List	200	400	40	41.500
List	400	400	40	75.100
Tree	1	400	40	39.000
Tree	2	400	40	23.350
Tree	8	400	40	16.400
Tree	10	400	40	17.650
Tree	20	400	40	20.600
Tree	40	400	40	21.700
Tree	100	400	40	26.600
Tree	200	400	40	39.050
Tree	400	400	40	68.750

Table 2.2

As it can be seen from table 2.2 the data gather by the tests do not confirm our second hypothesis on the data structure performance behavior by increasing the number of threads (it confirms again our first hypothesis since the tree data structure always runs faster than the list one when having the same parameters). The list performance behavior and the tree one can be seen respectively in the chart 2.3 and chart 2.4. As we can see from the tables and the charts the two data structures are behaving exactly the same: after incrementing from 8 to 10 threads the execution time increases. At a first glance this may seem not logical, but subsequent runs with an increased number of threads result in slower execution. There is a clear cutoff in performance between 8 and 9 threads (this is valid for this testing session where we used these exact values, with other parameters the cutoff could be between a different number of threads). In theory this happens because of different problems that are common in parallel programming:

- **Creating** and **destroying** each thread adds load to the total overhead, hence the more the threads the more the program overhead
- Saving and restoring a **thread register** state adds load to the total overhead.
- Saving and restoring the **cache state** of a thread adds load to the total overhead.

- The more the threads the more probable is that **virtual memory** will be **trashed**. Retrieving memory from the disk is a very expensive operation that adds load to the total overhead.
- **Lock convoy** can happen when the implemented programs deal with coarse grained locking data structures. When multiple threads with the same priority want to access the same data structure and fail in acquiring the lock they add overhead to the program execution. The overhead consist in the repeated context switches from one thread to another when a lock acquaintance is failed.

To see which one of the previous points affected our program performance we should run again the programs under the linux command line tool perf. This analyzes the execution of the program and it saves in a report (that can be accesses after the `perf record your_program` command with the `perf report` one). We did four different runs with perf and analyzes the report content. We first run the list data structure with 8 threads, 400 items and 40 as work time. After this run we did the same but increasing the number of threads from 8 to 400 in order to have a much different work load to analyze (running with 400 threads emphasizes why running the program with more threads slows it down). The third and fourth run were the same as the first and the second ones but with the tree data structure. The tests were performed on a local machine due to the impossibility of accessing the perf utility under the DAS4 node environment. As it can be seen from the first and the second reports (see image 2.5 and 2.6) and from the third and the fourth one (see image 2.7 and 2.8) increasing the number of threads makes the program dealing most of the time with this call: `rwsem_down_write_failed`. This call happens when a thread does not succeed to acquire a lock

perf record ./bin/test_data_structures cgl 8 400 40

Percentage work	Caller	Library	Prefix	Call
2.01%	java	libz.so.1.2.8	[.]	inflate
1.87%	java	[kernel.kallsyms]	[k]	copy_user_enhanced_fast_string
1.58%	java	perf-11779.map	[.]	0x00007f017c7e5d85
1.40%	java	perf-11779.map	[.]	0x00007f017c7da4c8
1.33%	java	libjvm.so	[.]	0x00000000008f736c

Image 2.5

perf record ./bin/test_data_structures cgl 400 400 40

Percentage work	Caller	Library	Prefix	Call
2.69%	java	[kernel.kallsyms]	[k]	rwsem_down_write_failed
2.17%	java	[kernel.kallsyms]	[k]	_raw_spin_lock
1.54%	java	libz.so.1.2.8	[.]	inflate
1.01%	java	libjvm.so	[.]	0x00000000008f736c
0.92%	java	[kernel.kallsyms]	[k]	clear_page_c_e

Image 2.6

perf record ./bin/test_data_structures cgt 8 400 40:

Percentage work	Caller	Library	Prefix	Call
1.72%	java	libz.so.1.2.8	[.]	inflate
1.69%	java	perf-12409.map	[.]	0x00007f7c0d0124f5
1.51%	java	libjvm.so	[.]	0x00000000007c44af
1.44%	java	libjvm.so	[.]	0x00000000007c4784
1.43%	java	[kernel.kallsyms]	[k]	_raw_spin_lock_irqsave

Image 2.7

for writing to the data structure. On the other hand with 8 threads the most expensive call was the one dealing with data decompression (`inflate`). It is clear, from the gathered data, that the problem why the program runs slower if it is executed with a number of threads bigger than 10 it is lock convey. Multiple threads, in fact, try to acquire a lock in order to write the items to the data structure but they fail since there is already another thread having the lock. The context switch after failing to acquiring the lock causes a lot of overhead. This is the cause of the biggest overhead for both the data structure. It can also be noted that the tree data structure is slowed down by the `page_fault` (see second entry image 2.8) call that is the consequence of the virtual memory trashing (this probably because the tree data structure performs its operation in a recursive way creating many memory instances). The list, on the other hand, it is slowed down by the `_raw_spin_lock` call (see second entry image 2.6) that means that the threads are repeatedly trying to access the wanted resource in a loop.

perf record ./bin/test_data_structures cgt 400 400 40

Percentage work	Caller	Library	Prefix	Call
4.87%	java	[kernel.kallsyms]	[k]	rwsem_down_write_failed
1.88%	java	[kernel.kallsyms]	[k]	page_fault
1.42%	java	libjvm.so	[.]	0x00000000008f736c
1.32%	java	[kernel.kallsyms]	[k]	_raw_spin_lock
1.19%	java	libc-2.19.so	[.]	malloc

Image 2.8

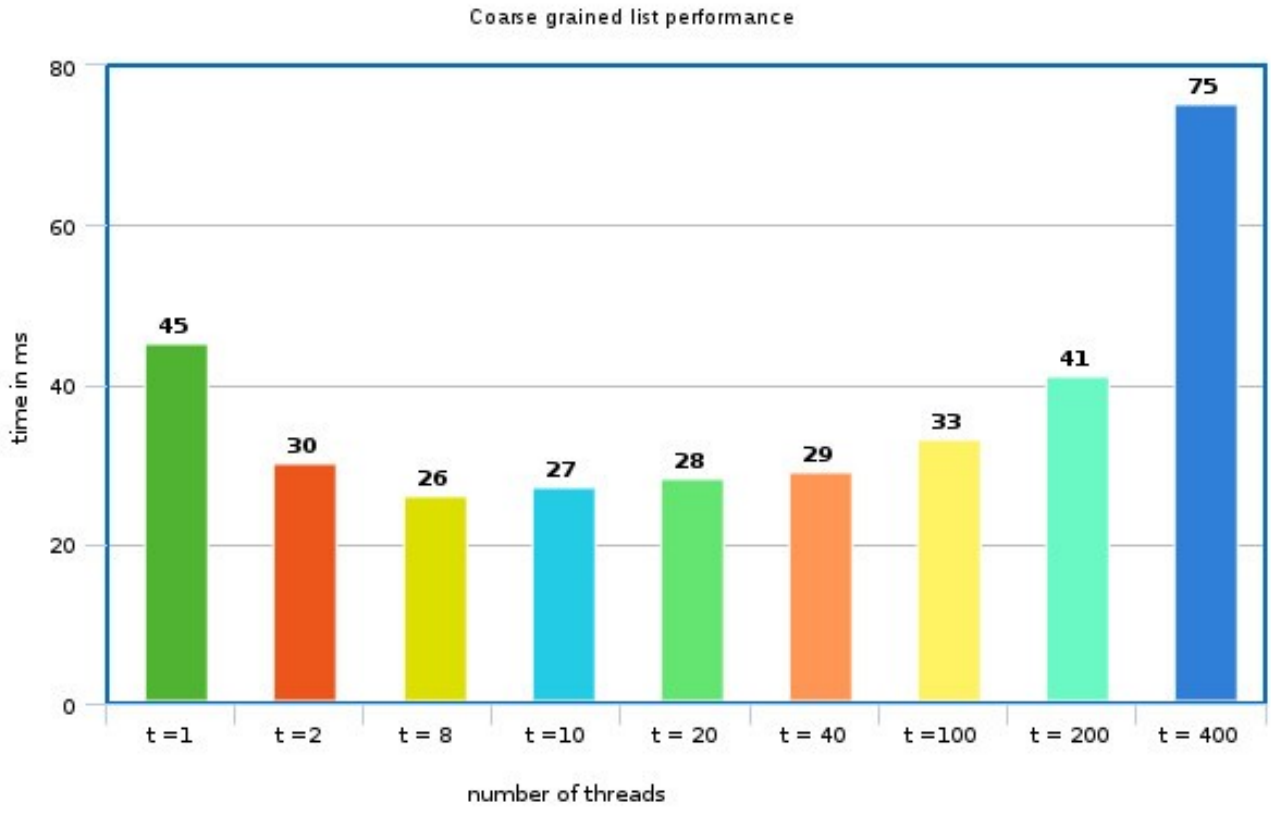

Chart 2.3

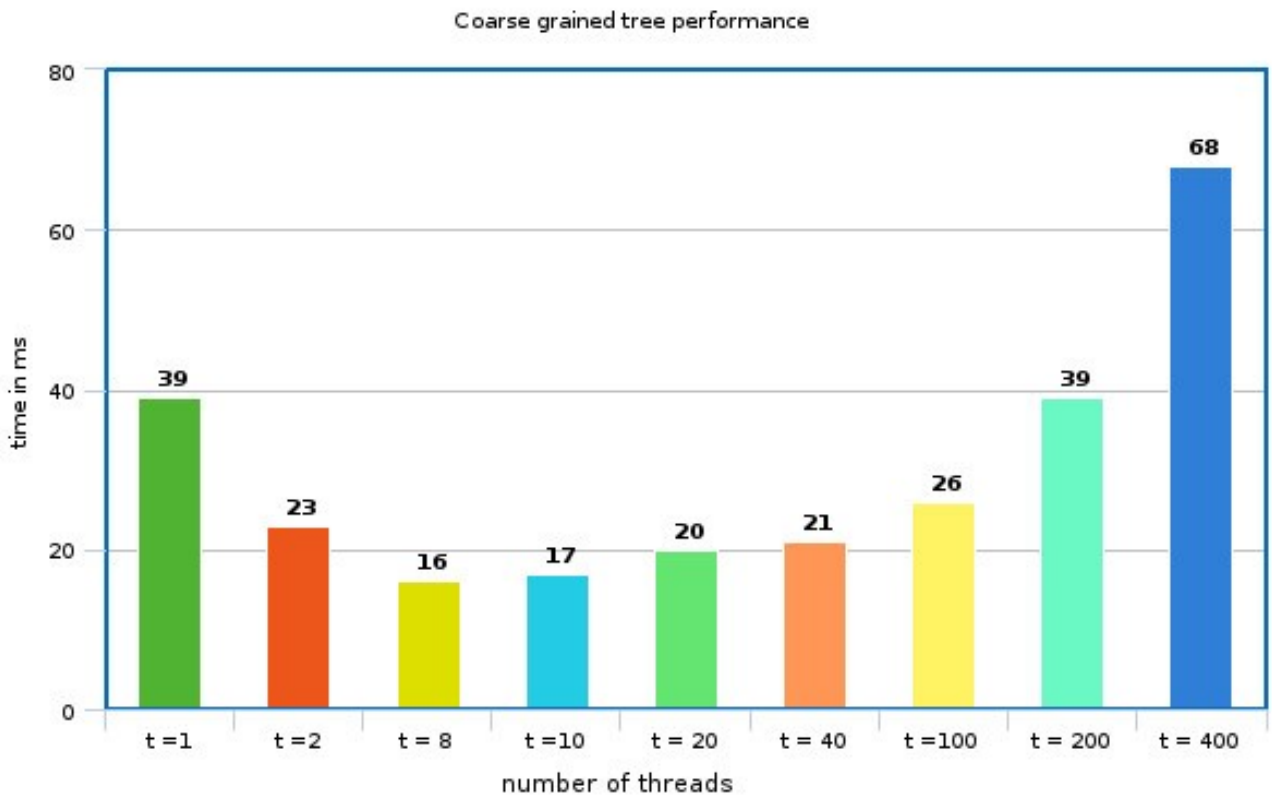

Chart 2.4

Appendix A:

The following script runs the input test data structure program 20 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 $(seq 1 20)
do
    y=$($command | grep time | awk '{print $2}')
    x=$((x+y))
done
average=$(echo "$x/20" | bc -l)
echo "for the command $command the average is: $average"
```