Lab 3 - Validating linearizability of Lock-Free SkipLists

October 15, 2023

- Group 2
- de Sotto, Diego

Measurements are taking using the Measurement class. Code for it can be found in the src/Measurement.java file. Execution and compilation where performed as follows:

File 1 `javac -d ./bin src/Measurement.java`

File 2 `java -cp ./bin Measurement [logMethod] [type] [threads] [numOps] [max]`

Arguments are explained as follows :

```
* logMethod - whether to use global lock, local lock or a lock-free
  pre-implemented structure.  Options are "basic" (no logging),
  "globlock" (locked global log), "loclock" (thread specific log)
  and "nolock" (thread safe log)

* type - thread operations (A1, A2, B1, B2) as referenced in instructions

* threads - number of threads to be used

* numOps - number of operations per thread

* max - maximuma value to be sampled
```

1 Task 1 : Measuring execution time

1.1 - Measurement program

Relevant files:

File 1 `scripts/localTime.sh` (Bash script that runs the Measurement program with 100 000 operations per thread with 1,2,4, 8 threads and outputs the results to the data directory with filenames 'localExectimetype.dat', where type references the operations performed by the threads).

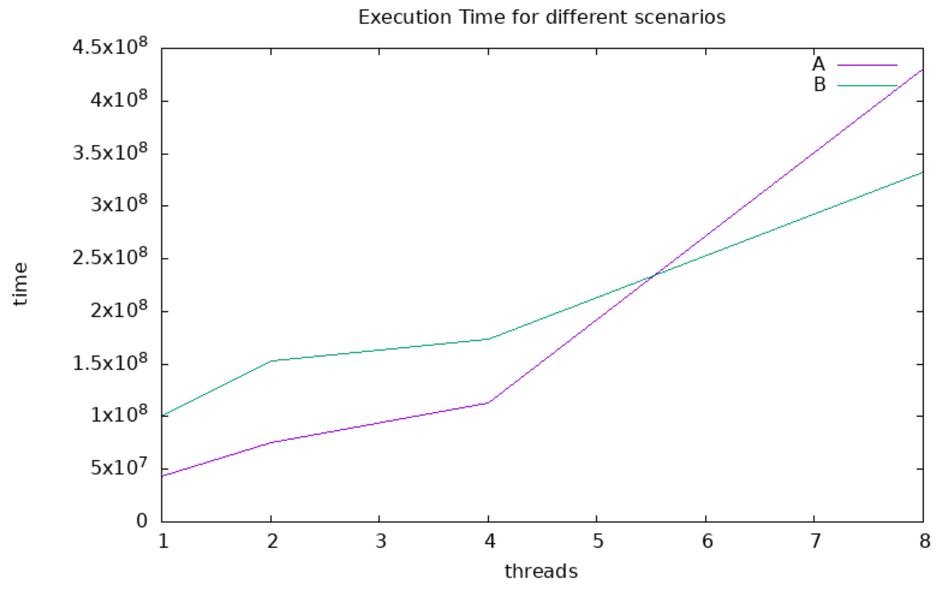

Figure 1: Local Execution

1.2 - Dardel experiments

Relevant files:

File 1 `scripts/pdcTime.sh` (Bash script that runs the Measurement program with 1 000 000 operations per thread with 1,2,4, 8, 16, 32, 64 and 96 threads and outputs the results to the data directory with filenames 'localExectimetype.dat', where type references the operations performed by the threads).

As can be seen in the plot, the execution time goes up as the thread count increases. Although this may raise concerns at first sight, after a little reasoning one realises that the higher the number of threads, the higher the contention for writing/reading to memory. What's more, since each thread executes a predetermined number of operatioins, the operationi count increases linearly with the number of threads (total num ops = threads * num ops per thread).

2 Task 2 : Identify and validate linearization points

2.1 - Identify linearization points

* `add()` -

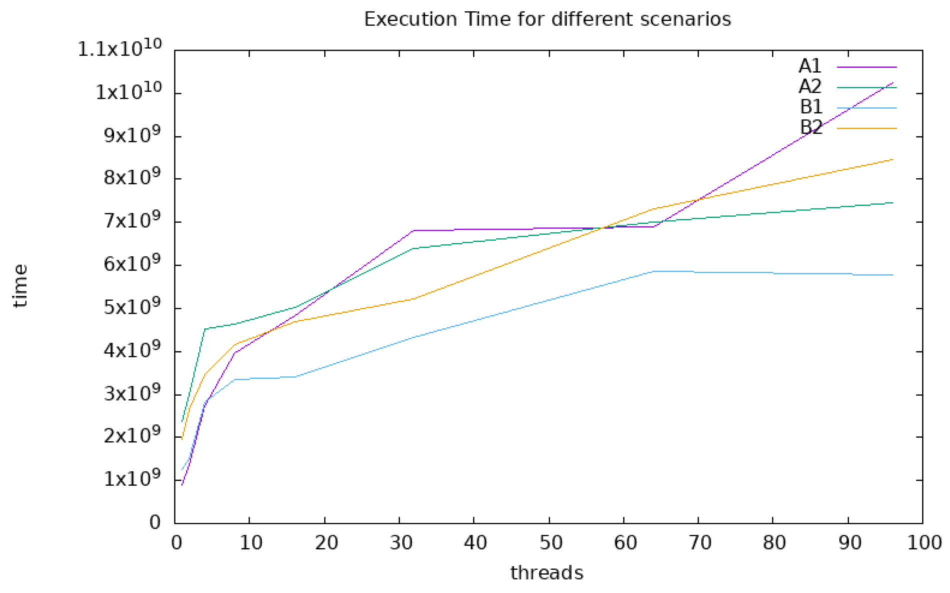

Figure 2: PDC Execution

- **return true** - in the CAS call to set newNode as successor in bottom level. If it returns true (if condtion is false), node is added. Otherwise, keep looping. Log is introduced right after
- **return false** - in the find() call if it returns true (element already in list)

* `remove()` -

- **return true** - after node is marked as logically removed using the CAS call when defininf iMarkedIt variable - can be recorded inside if iMarkedIt statement if true
- **return false** - in the find() method call if it returns false (element not in list)

* `contains()` - since the contains() method does not modify the list, the linearization point is the read of the curr variable when determining the boolean for the function return

2.2 - Develop a validation method

Relevant files:

File 1 `src/Log.java` (Implementation of validation method using switch block and HashSet, as well as method call registry in an Entry class).

2.3 - Locked time sampling

Relevant files:

File 1 `src/globLockLockFreeSkipList.java` (Use ArrayList as log and synchronized blogs to ensure exclusive read and write to array).

2.4 - Lock-fee time sampling with local log

Relevant files:

File 1 `src/locLockLockFreeSkipList.java` (Implement ArrayList log for each task, then merge by timestamp into common log).

2.5 - Lock-free Time Sampling with Global Log

File 1 `src/noLockSkipList.java` (Use a CopyOnWriteArrayList as a log. CopyOnWriteArrayList is thread safe. More info on here.)

2.6 - Dardel experiments

2.6.1 Global Lock

Execution time increases quite linearly, which suggests it scales with number of operations - fixed number of operations per thread means higher thread count involves more method calls.

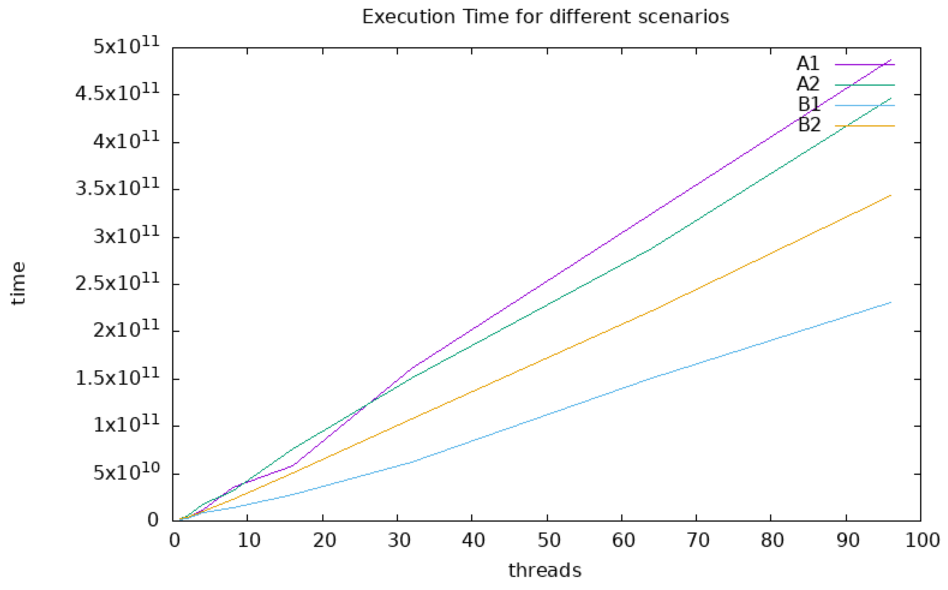

Figure 3: Global Lock execution

2.6.2 Local Lock

Surprisingly, execution time does not behave in the same linear fashion as it did with the locked global sampling. The curve looks more like a logarithmic one, which suggests better performance for higher thread counts. What's more, execution time is two orders of magnitude less.

Although discrepancies in sampling start to appear for higher thread counts, the accuracy remains perfect (accuracy is measured as discrepancies / (number of ops * threads)).

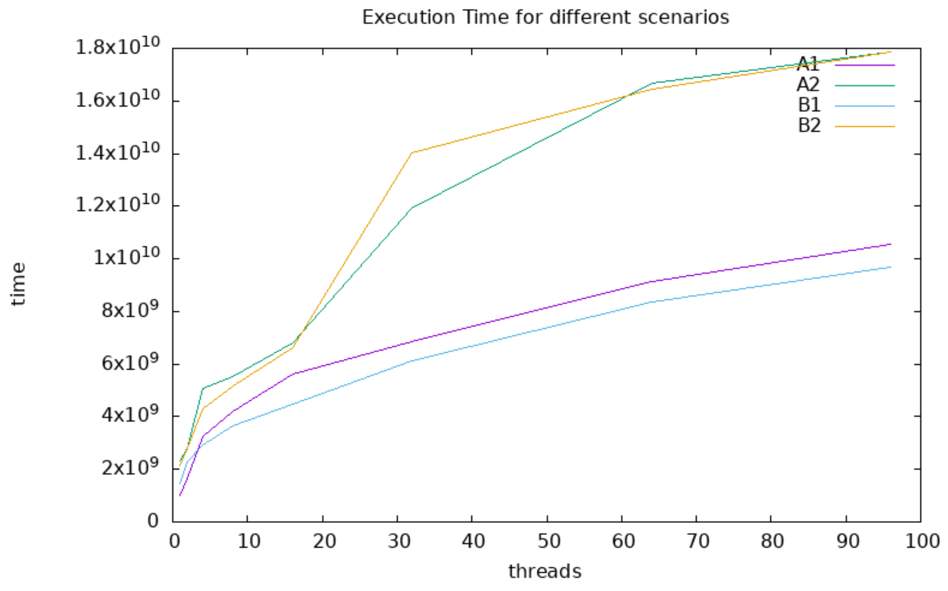

Figure 4: Thread specific log execution

	A1		**A2**		**B1**		**B2**	
1	0	1.00	0	1.00	0	1.00	0	1.00
2	0	1.00	0	1.00	0	1.00	0	1.00
4	0	1.00	2	1.00	12	1.00	2	1.00
8	0	1.00	2	1.00	10	1.00	18	1.00
16	1	1.00	2	1.00	19	1.00	112	1.00
32	12	1.00	26	1.00	187	1.00	72	1.00
64	81	1.00	78	1.00	736	1.00	1116	1.00
96	226	1.00	250	1.00	1848	1.00	3304	1.00

Table 1: Accuracy data for Local Lock

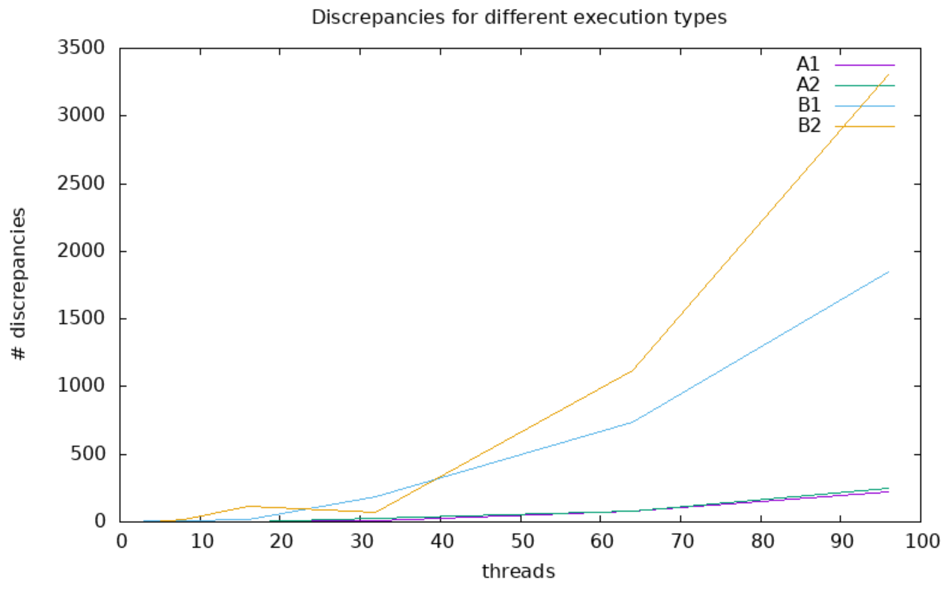

Figure 5: Discrepancies for thread specific log execution

2.6.3 Lock-free log structure

Once again, we return to linear behaviour. Execution time increases once again to numbers seen in the locked global logging implementation. However, the biggest thing to point out is there is not much of a difference between execution types, which could be appreciated in the two previous implementations.

The number of discrepancies is quite higher for increasing thread counts, with the behaviour remaining similar to the exponential curve seen in the local log. Accuracy still remains perfect, considering the number of discrepancies is still pretty low compared with the number of method calls performed.

	A1		**A2**		**B1**		**B2**	
1	0	1.00	0	1.00	0	1.00	0	1.00
2	0	1.00	0	1.00	0	1.00	0	1.00
4	0	1.00	2	1.00	0	1.00	6	1.00
8	0	1.00	0	1.00	7	1.00	8	1.00
16	11	1.00	80	1.00	241	1.00	984	1.00
32	74	1.00	586	1.00	1141	1.00	5863	1.00
64	517	1.00	2768	1.00	5294	1.00	17285	1.00
96	1231	1.00	6510	1.00	12045	1.00	61516	1.00

Table 2: Accuracy data for No Lock

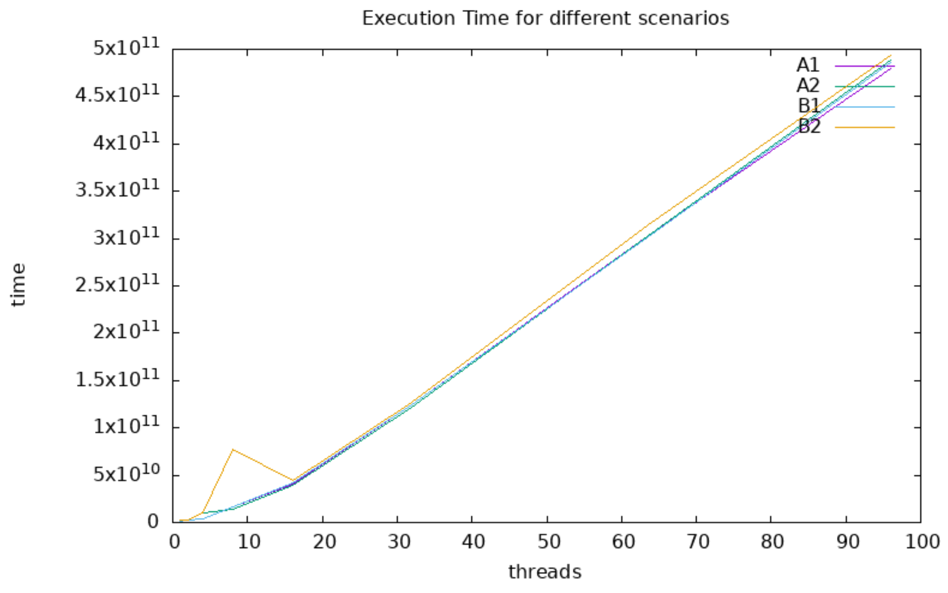

Figure 6: Thread safe log execution

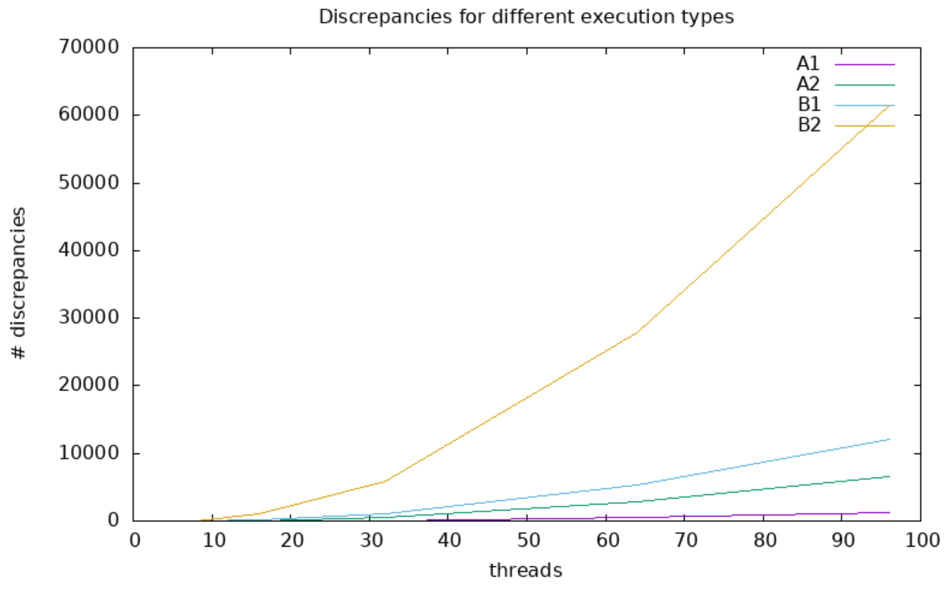

Figure 7: Discrepancies for thread safe execution