SLR 206 - Concurrent list based set: performance analysis.

Gélébart Mael

Guittard Adrien

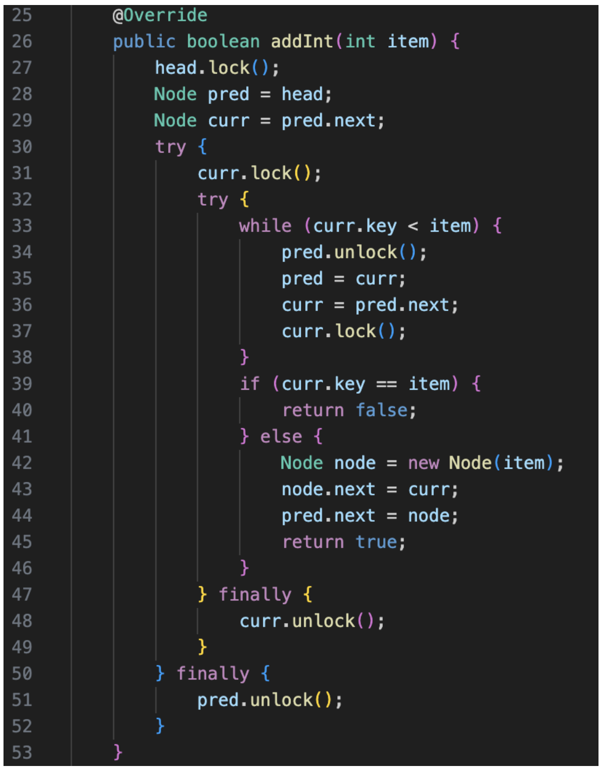
Correctness Hand Over Hand

In this part, we will discuss the correctness of the hand over hand algorithm proposed in our project.

Liveness:
For all the methods implemented, we are using starvation free locks and the method returns true or false after a finite number of steps, hence the starvation freedom is assured as long as there are no deadlocks. The second assumption results from the fact that there are no possible infinite loops in the code. In general, deadlock freedom and starvation free locks do not assure the starvation freedom. The deadlock freedom is elementary, one process enters the critical section by acquiring a lock and releases it right after. There is no possible situation resulting in a deadlock.

Safety:
We shall prove that the methods implemented are linearizable. We will show that this property holds for each one of them.

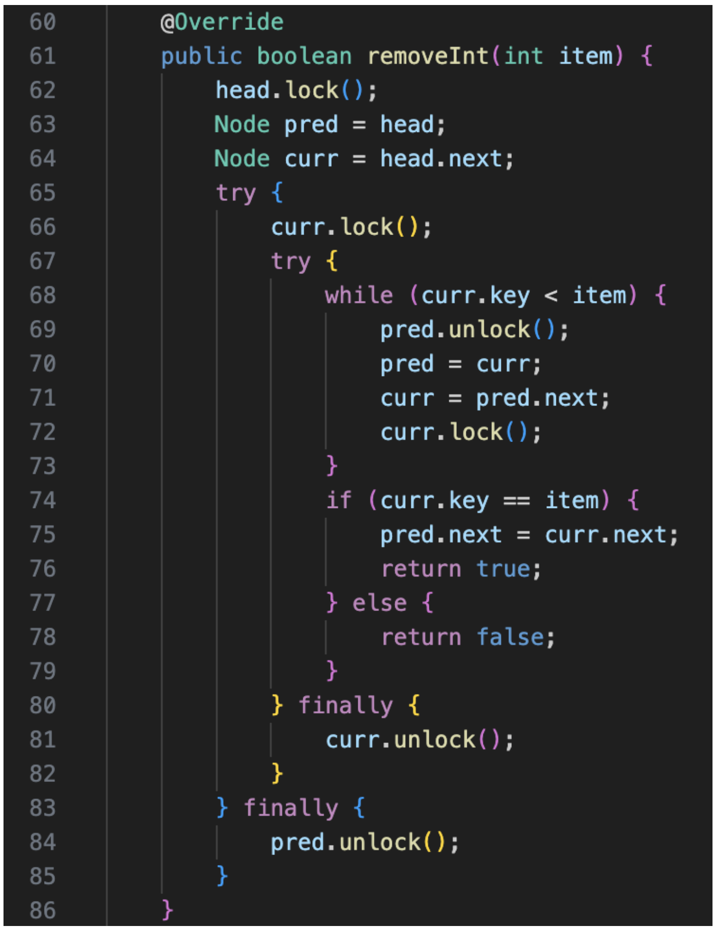
For the update operations, during the travers of the list, we successively lock two adjacent nodes “curr” and “pred” until reaching the desired nodes. This travers is done in a way that no other process can access the nodes “curr” and “pred” while the current process is performing the update.



```
25      @Override
26      public boolean addInt(int item) {
27          head.lock();
28          Node pred = head;
29          Node curr = pred.next;
30          try {
31              curr.lock();
32              try {
33                  while (curr.key < item) {
34                      pred.unlock();
35                      pred = curr;
36                      curr = pred.next;
37                      curr.lock();
38                  }
39                  if (curr.key == item) {
40                      return false;
41                  } else {
42                      Node node = new Node(item);
43                      node.next = curr;
44                      pred.next = node;
45                      return true;
46                  }
47              } finally {
48                  curr.unlock();
49              }
50          } finally {
51              pred.unlock();
52          }
53      }
```

Insert (addInt) :

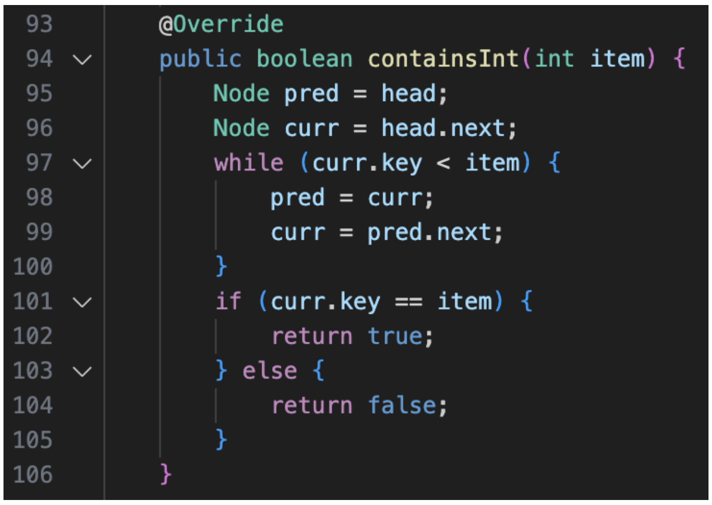
An insert is an update operation done in between the nodes “curr” and “pred”. The previous assumption is enough to prove that once the process acquires the locks, the two nodes considered are in fact in the list. If the insert returns false, the linearization point is at line 40. In the other case, the linearization point is at line 44. During the operation the state of the list between the locks is valid at any point, hence these linearization points only produce valid sequential histories.



```
60      @Override
61      public boolean removeInt(int item) {
62          head.lock();
63          Node pred = head;
64          Node curr = head.next;
65          try {
66              curr.lock();
67              try {
68                  while (curr.key < item) {
69                      pred.unlock();
70                      pred = curr;
71                      curr = pred.next;
72                      curr.lock();
73                  }
74                  if (curr.key == item) {
75                      pred.next = curr.next;
76                      return true;
77                  } else {
78                      return false;
79                  }
80              } finally {
81                  curr.unlock();
82              }
83          } finally {
84              pred.unlock();
85          }
86      }
```

Remove (removeInt) :

Here again, the travers ensures that the two nodes “curr” and “pred” are in the list at the moment the process acquires the locks. The way the updates are made, the next node after curr cannot be removed by any other process while the current one holds those nodes. This is enough to ensure that the state of the list remains valid and no illegal operation is made. The linearization points are at line 75 and 78 in this algorithm. It only produces legal sequential histories according to the argument above.



```
93      @Override
94      public boolean containsInt(int item) {
95          Node pred = head;
96          Node curr = head.next;
97          while (curr.key < item) {
98              pred = curr;
99              curr = pred.next;
100         }
101         if (curr.key == item) {
102             return true;
103         } else {
104             return false;
105         }
106     }
```

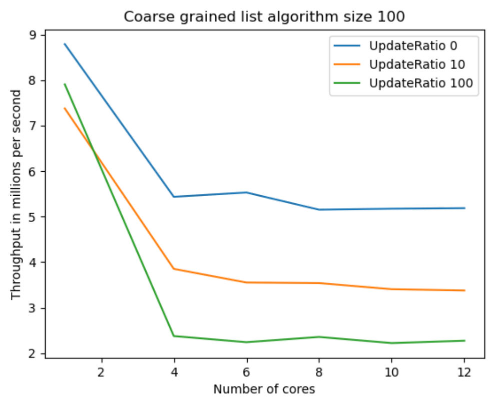
Contains (containsInt) :

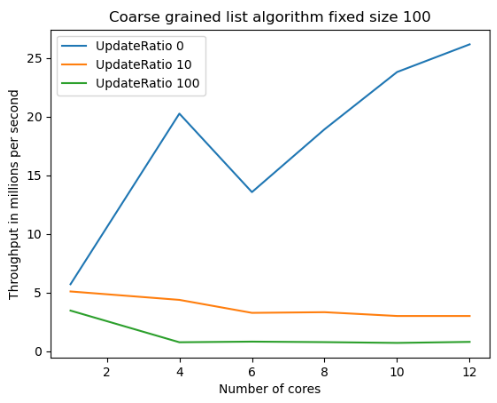
If the method returns false or true, the linearization point is, in either cases, at line 97, when the condition of the while clause becomes false. This will happen in a finite number of iterations because the list is supposed to be finite. This linearization point only generates valid sequential histories because at that moment, the method establishes that the element is or is not in the list (depending on the value of the current node key). If an update is occurring concurrently, the contains operation returns either the state of the list before the update is linearized or after depending on the case. This ensures that any history produced is a valid sequential history. Note that this might have not been the case if the lines 44 and 43 in the addInt operations were reversed (because there are no locks involved during the travers of the contains operation).

Performance analysis

For the performance analysis, we generated the graphs by extracting the raw data using a simple python script. The graphs were obtained using an AMD EPYC processor with 64 cores and 128 threads.

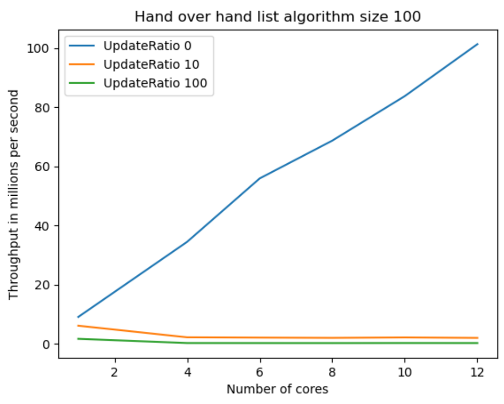
To start off, let's look at the three algorithms when we have a fixed list size of 100. For an update ratio of 0, (in blue), we should expect a linear progression of the performance for each added core if the contains doesn't use any locks. This is the case for hand over hand. Lazy linked list which performs ~8 million operations per second for each added core. The lazy linked implementation uses list.contains to perform the operation, which seems to behave differently at lower core counts, but we still end up with ~8 million operations per second per core at 12 cores. As for the coarse grain algorithm, the given implementation had a lock in the contains function. As such the performance is the same as the other 2 for 1 core but as soon as multiple cores are involved, the locks interfere and there seems to be a cap at a total of ~5 million operations per seconds no matter the core count. We also did a run of all the algorithms on a dual XEON CPUs (2* 6 cores, 12 threads) with no locks in the contains function of the coarse grain algorithm. The overall performance is lower due to the XEON being a much slower cpu but we see performances that scale with the cores (with an inconsistency for 6 cores, but we suspect that it is because of a hardware issue, perhaps the cache became the limiting factor if the 6 cores were on the same CPU).

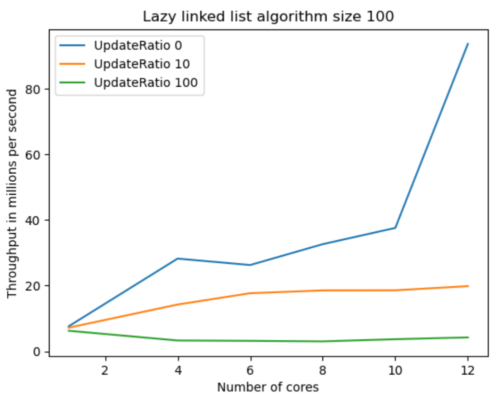




On the left the AMD EPYC simulation, on the right the intel XEON simulation

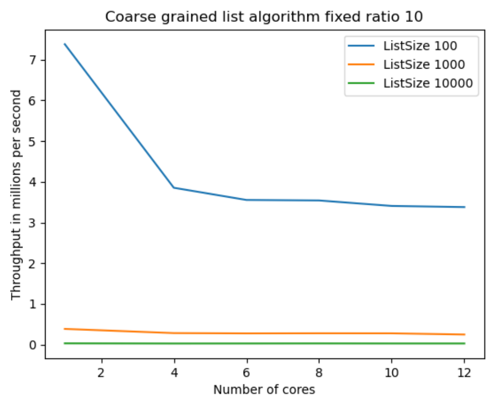
For the ratios of 10 and 100 (in orange and green), the coarse grain and the hand over hand algorithm have decreased performance as soon as multiple cores are involved. Adding cores is, at best, not affecting the performance. In many cases it even hinders performance. For the coarse grain with the contains using locks, the smaller yet noticeable dip in performance is attributable to the fact that the remove and add functions simply require more computing power. At 12 cores, they both hover at ~3 M operations per second for the 10% ratio and less than 1M for the 100% ratio. The lazy linked algorithm does much better at the 10% ratio, with 20M OPS/s but this cap is obtained at 6 cores, with no benefit for the added core beyond that. For a 100% update ratio the lazy linked list implementation only manages to do better (about 5 times the performance) with 1 core.

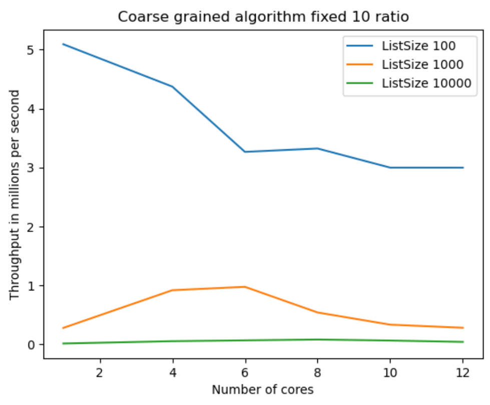




Both AMD EPYC simulations

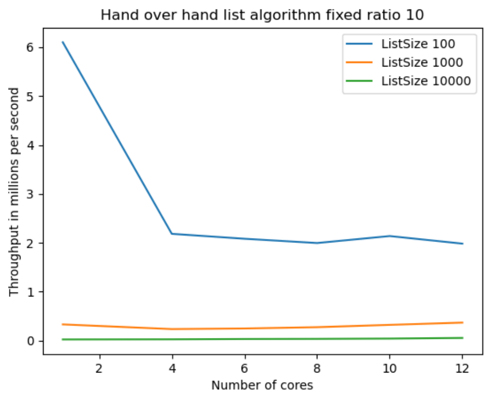
Moving on to the fixed 10% ratio, The coarse grained algorithm's performance gets killed as soon as the list size is 1000 or more (in green and orange), with less than 1M OPS/s. With a list size of 100 (in blue), the performance decreases with the number of cores as the locks interfere. However the performance seems to be capped at around 6 cores and doesn't decrease beyond that.

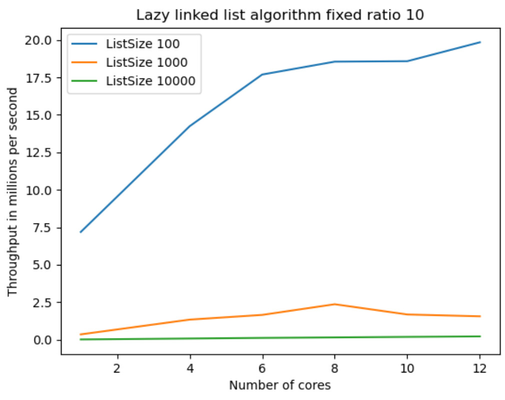




On the left the AMD EPYC simulation, on the right the intel XEON simulation

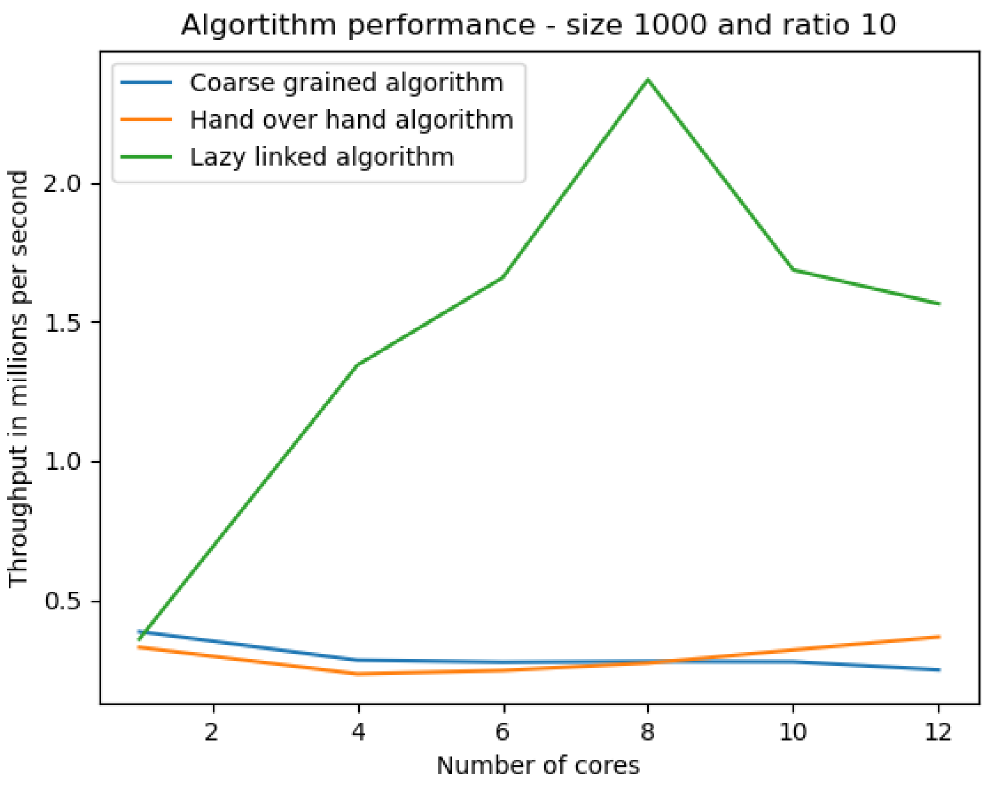
The hand over hand algorithm suffers a similar fate to the coarse grained one, virtually no performance with a list size of 1000 or more (in green and orange). For a size of 100 (in blue), we see a similar dip in performance with the first few added cores with a lower cap that seems to be reached at 6 cores. The lazy linked algorithm has the same performance as the other 2 for a list of 10 000. For size 1000, lazy linked seems to be actually benefiting from the additional cores and achieves a performance of ~2.5M OPS/s , again a cap to the performance seems to be reached at around 8 cores. The biggest difference with the other 2 algorithms is for the list size of 100, the performance increase is almost linear for the first few cores and seems to taper off but not before achieving ~20M OPS/s with 12 cores, about 15-20 the performance of the other 2.





Both AMD EPYC simulations

For size 1000 and 10% ratio, the coarse grained and the hand over hand algorithms have significantly worse performances than the lazy linked. With 8 cores and below, the coarse grained performs better than the hand over hand. With 10 or more cores, the trend reverses and the hand over hand has almost double the performance at 12 cores. This is what we would expect in theory, the hand over hand should perform better than the coarse grained when the list is sufficiently large and there are enough cores. Again the lazy linked algorithm tells a different story. Up until 8 cores, the performance gain is linear. With more than 8 cores the performance dips, this is not what we would expect from a theoretical standpoint. We assume it could be due to hardware limitations.



AMD EPYC simulation

Conclusion

Let's sum up the global performance of the 3 algorithms. First of all, for an update ratio of 100%, the performance is weak no matter the algorithm compared to the 10% ratios. For a list size of 10 000, the hand over hand performs about twice as well as the coarse grained. For an update ratio of 0, any algorithm will have comparable performances. Beyond that, the lazy linked outperforms the other 2 every single time for the 10% update ratio and the list sizes of 100 and 1000. If the list size is small enough, the coarse grained and hand over hand can actually be used and have decent performance. The lazy linked version of the algorithm is the one with the best scaling. It multiplies its performance with each added core.

In a sequential environment, the coarse grained would be recommended for its simplicity. The hand over hand only surpasses the coarse grained with added cores and a sufficiently large list size. Finally if starvation freedom is not a concern, the lazy linked list is the best, from those three, especially in a highly parallelized environment since it has the best scalability by far.