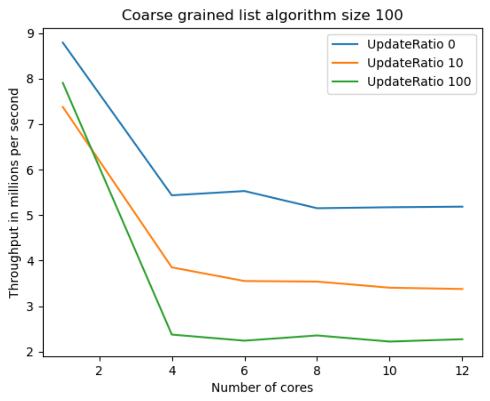
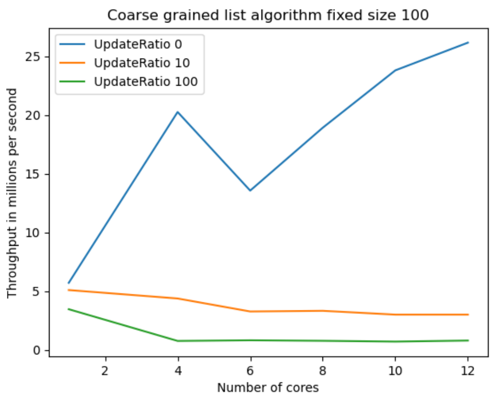
Gélébart Mael
Guittard Adrien

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.

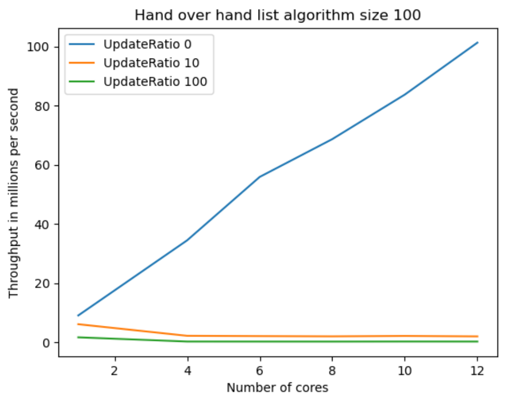
To start off, let’s look at the three algorithms when we have a fixed list size of 100. For an update ratio of 0, (in blue), we should expect a linear progression of the performance for each added core if the contains doesn’t use any locks. This is the case for hand over hand. Lazy linked list which 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).

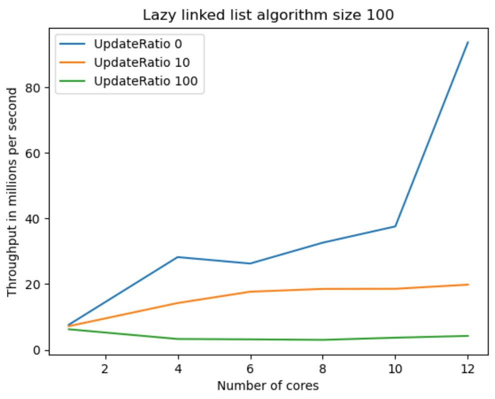




On the left the AMD EPYC simulation, on the right the intel xeon simulation

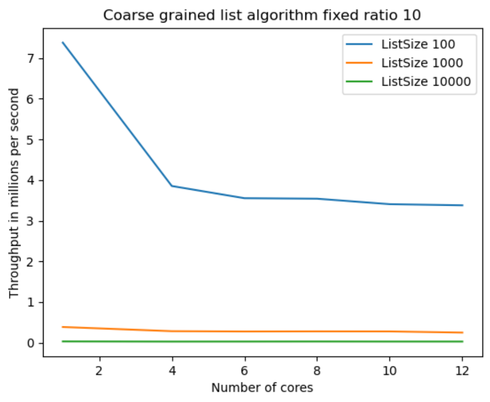
For the ratios of 10 and 100 (in orange and green), the coarse grain and the hand over hand algorithm have atrocious performance as soon as multiple cores are involved. Adding cores is, at best, not affecting the performance. In many cases it even hinders performance. For the coarse grain with the contains 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 ~5 M operations per second for the 10% ratio and ~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.

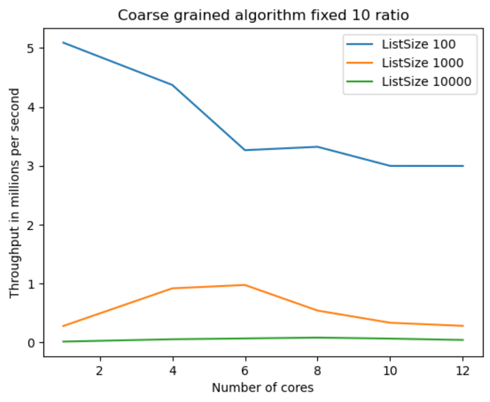




Both AMD EPYC simulations

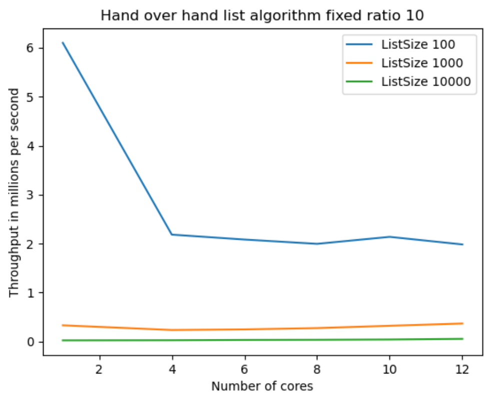
Moving on to the fixed 10% ratio, The coarse grained algorithm's performance gets killed as soon as the list size is 1000 or more (in green and orange), with less than 1M OPS/s. With a list size of 100 (in blue), the performance decreases with the number of cores as the locks interfere. However the performance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

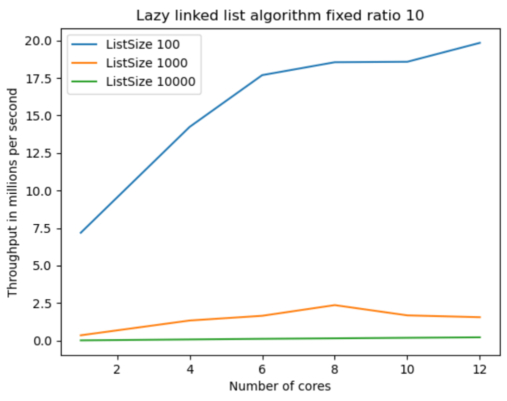




On the left the AMD EPYC simulation, on the right the intel XEON simulation

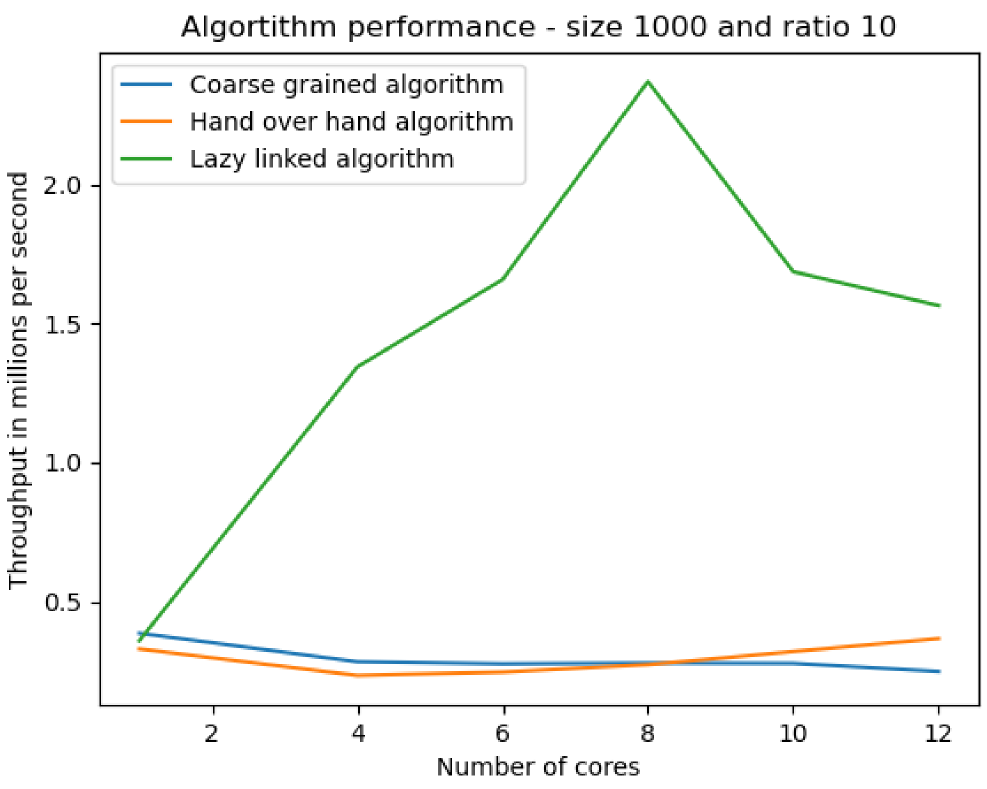
The hand over hand algorithm suffers a similar fate to the coarse grained one, virtually nor performance with a list size of 1000 or more (in green and orange). For a size of 100 (in blue), we see a similar dip in performance with the first few added cores with a lower cap that seems to be reached at 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 indiscernible performance. Unfortunately, they both have the same underwhelming results. The added cores seem to do absolutely nothing and the variation in performance is probably due to hardware variance. 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, probably because there are enough cores and the locks start interfering with each other.



AMD EPYC simulation

Let's sum up the global performance of the 3 algorithms. First of all, for a list size of 10 000 or an update ratio of 100%, the performance is horrible no matter the algorithm. For an update ratio of 0, any algorithm will do. 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. It seems that if the list size gets too large or if the update ratio gets too big, only lazy linked maintains good performance and actually scales with added cores.