In this project I optimized knapsack and parallelized knapsack with coarse and fine grain parallelization. For my testing I made a custom 50x2million input file from part of the input of the 100x20M file for time efficiency of generating graphs. All experiments were run on the phoenix machine.

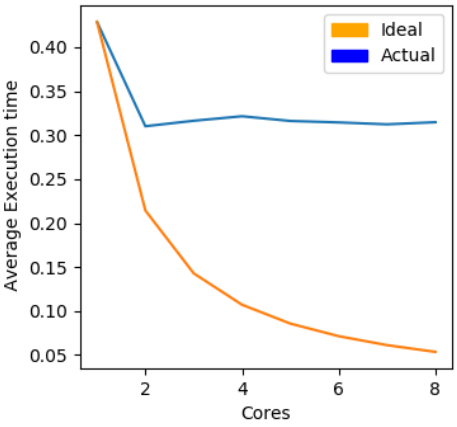
Also, the I am getting speedups that should be correct when I test my code on the machine, but the autograder has values that are pretty far off of it.

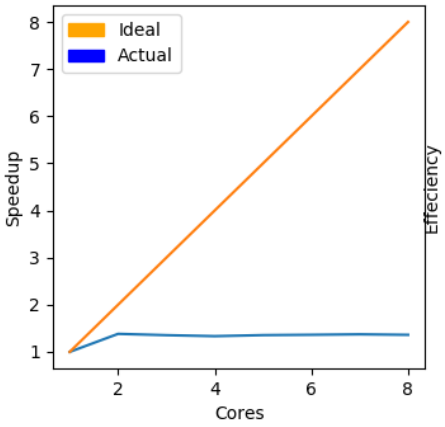
Here are the speed ups of all my programs vs knapSeq on the biggest file we have

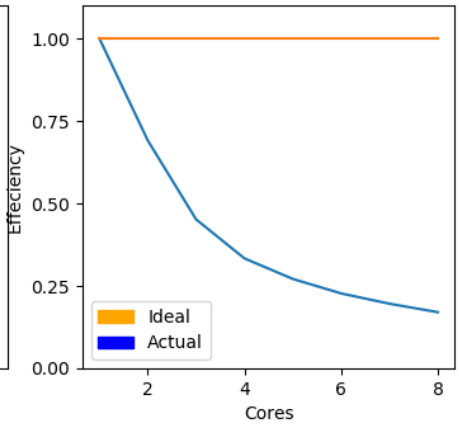
```
phoenix:~/Desktop/cs475/proj/P3.1/PA3$ knapSeq sample/k100x20M.txt
The number of objects is 100, and the capacity is 20000000.
The optimal profit is 43635511
Time taken : 8.489496.

phoenix:~/Desktop/cs475/proj/P3.1/PA3$ knap1 sample/k100x20M.txt
The number of objects is 100, and the capacity is 20000000.
The optimal profit is 43635511
Time taken : 9.412774.

Speed up = 0.90

phoenix:~/Desktop/cs475/proj/P3.1/PA3$ knap2 sample/k100x20M.txt 15
The number of objects is 100, and the capacity is 20000000.
The optimal profit is 43635511
Time taken : 6.139773.

Speed up = 1.38

phoenix:~/Desktop/cs475/proj/P3.1/PA3$ knap3 sample/k100x20M.txt
The number of objects is 100, and the capacity is 20000000.
The optimal profit is 43635511
Time taken : 3.566202.

Speed up = 2.38
```

My graphs follow. Since knap2, and knap3 run on 1 thread are practically close to identical to knap1 with no non negligible speedup from any changes, I use that the 1 core run as the base line to calculate speedup and efficiency compared to knap1.

Knap 2 with depth of 2







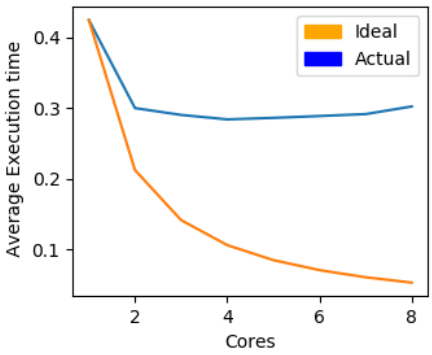
1	0.42864500000000005
2	0.3100815714285714
3	0.31631985714285715
4	0.32143057142857145
5	0.31607599999999997
6	0.31441199999999997
7	0.3122688571428571
8	0.314654

1	1.0
2	1.3823620604900742
3	1.3550998785587285
4	1.3335539245533583
5	1.3561453574456779
6	1.363322646718319
7	1.372679312058016
8	1.362274116966573

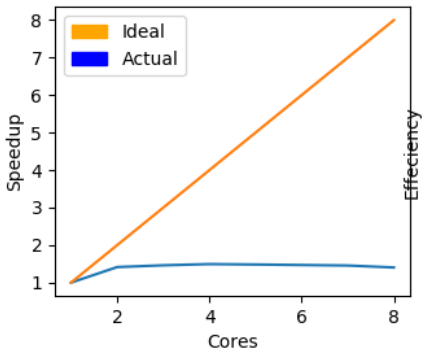
1	1.0
2	0.6911810302450371
3	0.45169995951957614
4	0.3333884811383396
5	0.27122907148913555
6	0.22722044111971984
7	0.19609704457971658
8	0.17028426462082163

\

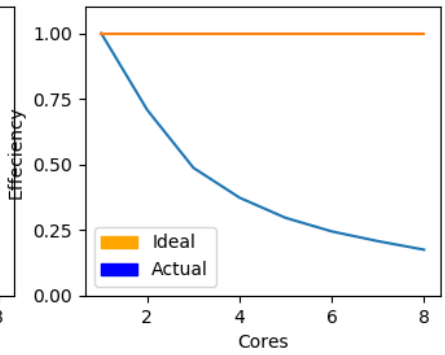
knap 2 with depth of 5



1	0.42431642857142854
2	0.299897
3	0.29024042857142857
4	0.2839395714285714
5	0.28605328571428573
6	0.28868885714285714
7	0.29140514285714286
8	0.3022041428571428

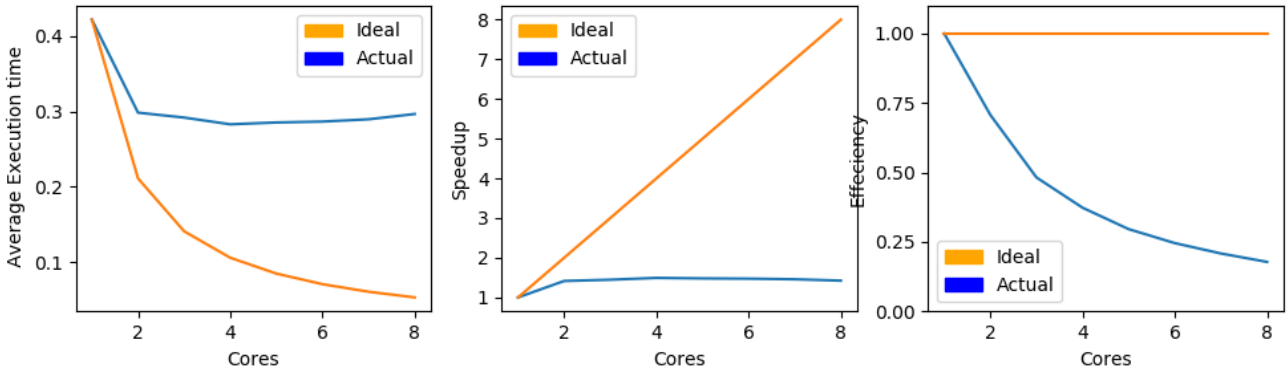


1	1.0
2	1.4148738685996476
3	1.4619480499664563
4	1.49438990288175
5	1.4833475081815424
6	1.4698053564341638
7	1.4561048045038913
8	1.4040721763765176



1	1.0
2	0.7074369342998238
3	0.48731601665548546
4	0.3735974757204375
5	0.29666950163630845
6	0.24496755940569395
7	0.20801497207198447
8	0.1755090220470647

knap2 with depth of 15



1	0.422047
2	0.29840485714285714
3	0.29186014285714285
4	0.28289971428571425
5	0.2853482857142857
6	0.2865402857142857
7	0.28958200000000006
8	0.296617

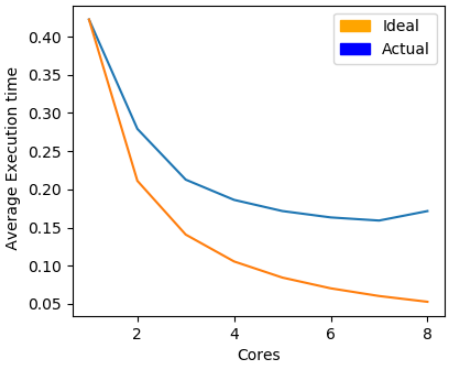
1	1.0
2	1.414343600305242
3	1.4460590468722545
4	1.4918608209471507
5	1.4790591748029225
6	1.4729063278063121
7	1.4574351997016386
8	1.422868547655731

1	1.0
2	0.707171800152621
3	0.4820196822907515
4	0.37296520523678767
5	0.2958118349605845
6	0.24548438796771868
7	0.2082050285288055
8	0.17785856845696638

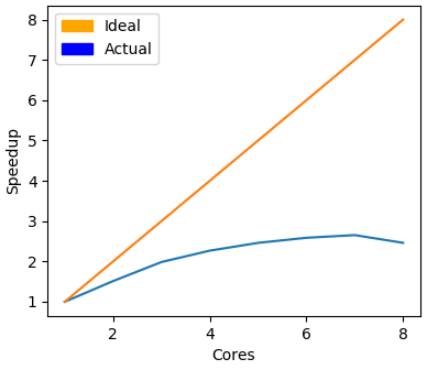
With a depth of 15, all calls will spawn tasks all the way down to the base case for an object set of size 50. This seems to be optimal for this small input size of all the numbers I tested, but I would assume that with an input with a massive number of objects, the optimal depth would probably be between 15 and 20. Since the max depth of the algorithm scales with log(n), 15 to 20 would go pretty far down the call tree for a big input.

Knap 2 gets some benefit from parallelization at first when it can use another core, but stagnates quickly. This makes sense because a lot of the work is in the equivalent function of getLastRow (mine has a different name), so it should have trouble getting much more performance.

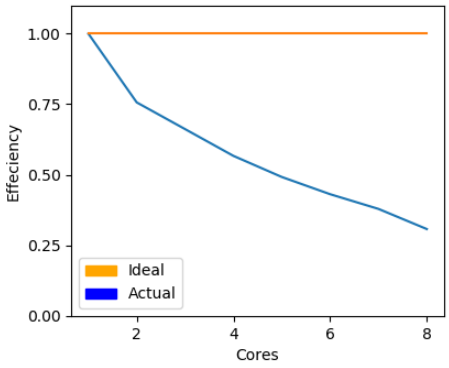
Knap3



1	0.4223204285714286
2	0.27916657142857143
3	0.21270614285714284
4	0.1862157142857143
5	0.17158942857142856
6	0.16323414285714286
7	0.15915357142857142
8	0.1715485714285714



1	1.0
2	1.5127901109731723
3	1.985464184995665
4	2.2679097206772485
5	2.4612263825776814
6	2.5872064580326772
7	2.653540381033593
8	2.461812564538157



1	1.0
2	0.7563950554865861
3	0.661821394998555
4	0.5669774301693121
5	0.4922452765155363
6	0.4312010763387795
7	0.3790771972905133
8	0.30772657056726965

The fine grain parallelization in knap3 works pretty well. In a lot of ways its still far off from ideal, but since the algorithm has a few parts that cannot be parallelized, that makes sense and we should expect less than perfect. The parallelization done was in the for loop of the getlastrow function, (even though mine is named solvesubkp).

The fine grain knap3 performed much better than the coarse grain knap2 parallelization since a large portion of the work is in the first call of getlastrow.