Jacob Barzee

P1: Parallelize CNN

Loop	Notes & File	Case	Seq Time	Par Time	Speed Up
N	schedule(dynamic)	1	5.904755	0.394681	14.960823
	cnn.par1.c	2	7.762375	0.719105	10.794492
N	schedule(static, 8)	1	6.233746	0.702076	8.879025
	cnn.par2.c	2	7.321761	1.073063	6.823236
K	schedule(dynamic)	1	5.799878	0.531019	10.922166
	cnn.par3.c	2	8.697852	3.410715	2.550155
N & K	schedule(dynamic) - both	1	5.703278	0.425945	13.389705
	cnn.par4.c	2	8.038259	0.752818	10.677556
P	schedule(static, 8)	1	10.73327	2.622695	4.092460
	cnn.par5.c	2	7.268169	210.51354	0.034526
P	schedule(static, 32)	1	5.504790	8.592416	0.640657
	cnn.par6.c	2	7.259249	208.53393	0.034811
C	schedule(dynamic) ordered	1	6.070164	24.968036	0.243117
	cnn.par7.c	2	7.313388	33.364343	0.219198
C	schedule(dynamic) critical	1	6.744068	72.476353	0.093052
	cnn.par8.c	2	Failed	Failed	Failed
N & K	schedule(dynamic) numthreads(4) - both	1	5.664050	1.764295	3.210375
	cnn.par9.c	2	7.280328	2.880403	2.527538
N & K	schedule(dynamic) numthreads(16 & 4)	1	5.954405	0.551465	10.797434
	cnn.par0.c	2	8.735133	1.051870	8.304384

Jacob Barzee

Explanations:

1. This was my fastest result. On a 28 core node this was roughly .5 efficiency which is ok. I think this one is fastest because there is so much work going on under any given thread. The outermost loop has enough iterations to consume all cores in parallel, and the cost of the parallelization is small in comparison to the parallel operations.
2. This test performed slower than #1, which is the same loop. I think this is because the scheduling choice ends up limiting the number of threads that can run in parallel. This wouldn't be the case if you had more total iterations to compete or if the block size was smaller.
3. This test performed similar to #1 in case 1 but not in case 2. I think this is because the inputs for case 2 minimize the number of iterations that the K loop goes through. That would also explain why #1,case 1 was comparable with this case 1.
4. This test was almost the same as #1. I think this is because the dynamic scheduling and openMP handled the situation well and didn't over build threads even though I had it working on both outermost loops. I expected this to perform worse, but now that I think about it I am not very surprised.
5. #5 performed exceptionally poorly for case 2. I think this is because I parallelize a loop that has only 3 iterations. (because of the input for case 2) Case 1 has ~30 iterations and thus performed far better. Despite performing better the efficiency is horrible, bordering on 25%. I was hoping to see some speed up here because of cache locality, but I think there is something I am missing.
6. With test #6 I tested my theory of the causes for #5. I think I am correct. When I raised the chunk size of the scheduling I essentially made it so that only one thread could operate on the loop (because the loop is the size of the chunk)
7. I tried to parallelize the C loop before but kept failing because of an output dependence, so I read up on the synchronization mechanism that openMP offers. The ordered command ensures that the ordered section runs in the same order that it would if the instructions weren't parallel. This essentially builds some intelligent synchronization mechanism which all threads will have to wait on. This can be effective parallelization if there is a good deal of overhead outside of the ordered section. I was hoping this would be the case with 4 loops between the parallelization and the ordered section, but I guess not.
8. I know that case #8 isn't guaranteed to find the correct solution every time, even though it did for case 1. I still included it because I think this is interesting that the critical command is enough to be fix it. I still don't know If I understand everything I would do to parallelize the C loop. I can't tell you for sure that it isn't possible either.

9. In case #9 I parallelized both outer loops but limited the number of threads that they had to work with. The efficiency up from this was mediocre, nearing 50%. I think that with this solution I had 8 threads total (4 + 4) but there is a chance that I had 16 (4 * 4).
10. To test my questions from #9 I tried the same solution, but gave the inner loop 16 threads. This saw substantially better performance. I am sure that “moar threads is moar better” but I am also curious if I was able to squeeze some more performance out of the fact that each outer thread had 4 inner threads to work with. I’m confident that there wasn’t any kind of thread dispatching that accounted for this, so it would have to be some sort of a byproduct. I also have a sneaking suspicion that the 4 outer threads spent the majority of their time waiting for the inner loop to complete so they could just step and start it again.