Chih Han Yeh

EC527

Lab 6

Spring 2025

Task 1

a.

```
char hola_world[] = "Hello World";
      int len = sizeof(hola_world) - 1;

  #pragma omp parallel for
for (int i =0; i < len; i++){
  printf("%c", hola_world[i]);
}

  printf("\n\n");
```

Output: lroWoHeldl

b. Comparing the output between OpenMP and the serial baseline with the set thread count as 4; once row length reaches 100 and beyond (from the output data), the overhead bound performance of the serial baseline becomes significantly slower than OpenMP, with computational times reaching 10 times of OpenMP (19.5ns to 1.95ns). Below is a table comparing the computational time vs row length of Overhead Bound Computation between the two methods, with threads of 1, 2, 4, and 8. As the table has shown, there are no performance differences when running on only one thread. 2 threads double the performance of OpenMP compared to the serial baseline, just like a polynomial equation, the performance of OpenMP increases with a power of two (2, 4, 8x etc) as thread count also goes up.

   Highlighted in yellow is the breakpoint where the performance improvements are significantly different between the two methods.

rowlen	Execution Time (ns)			
	1 Thread		2 Threads	
	ooverhead_bound_serial	omp_overhead_bound	ooverhead_bound_serial	omp_overhead_bound
16	0.287	0.961	0.266	1.104
20	0.138	0.372	0.15	0.461
28	0.299	0.409	0.327	0.6
40	0.548	0.681	0.599	0.79
56	1.275	1.247	1.222	1.006

76	1.987	2.357	2.436	1.466
100	3.866	3.291	3.541	2.199
128	6.415	5.636	6.247	3.278
160	9.231	25.240	9.638	5.182
196	15.310	13.410	16.07	7.326
236	39.320	19.600	23.08	29.94
280	31.610	49.640	63.9	15
328	43.870	55.860	60.53	27.75
380	89.790	50.400	70.78	27.4
436	133.100	66.390	93.79	41.1
496	149.000	107.500	139.8	83.9
560	183.400	177.600	119.2	62.62
628	216.700	177.000	153.6	108.1
700	225.200	197.600	225	162.1
776	280.700	261.000	278.6	141

rowlen	Execution Time (ns)			
	4 Threads		8 Threads	
	ooverhead_bound_serial	omp_overhead_bound	ooverhead_bound_serial	omp_overhead_bound
16	0.293	1.218	0.266	1.637
20	0.141	17.550	0.122	1.238
28	0.293	0.754	0.304	1.121
40	0.619	0.879	0.65	1.264
56	1.187	1.028	1.268	1.178
76	2.464	1.497	2.375	1.464
100	19.280	1.945	4.166	1.746
128	6.670	2.589	6.738	2.14
160	10.190	3.314	9.875	2.624
196	16.120	4.677	15.48	3.476
236	23.000	6.398	22.93	4.487
280	51.770	9.056	32.29	5.545
328	46.050	45.020	43.02	7.575
380	77.170	50.650	58.63	9.621
436	106.600	85.190	77.16	13.22
496	142.600	42.400	98.61	17.02
560	168.400	54.820	126.3	21.52
628	185.800	59.160	157.5	27.8
700	205.500	91.710	207	34.67
776	314.700	78.290	257.9	45.58

c.

Comparing the execution times between the two looping orders, order kij is faster than ijk, in orders of 2x. The C=computation time for ijk would reach 4s vs kij's 2s when the matrix size exceeds the L3

cache of 12MB. Comparing the serial baseline and OpenMP version, OpenMP could perform a speedup of 3.8x for the ijk loop ordering, and over 2.2x for kij. Placing pragma “parallel for” into the loops shows that being in the middle of the loop creates the fastest execution time, the outer of the loop being second, and the inner loop being the slowest. Also, placing the loop indices into the shared list displays small performance improvements with faster execution times. Below are tables comparing the performance of different OpenMP loop placements and loop indices placement.

Ijk vs kij No Modification

rowlen	ijk	ijk_omp	kij	kij_omp
64	0.000155	0.000134	0.000204	0.000285
112	0.001107	0.000367	0.000727	0.000468
192	0.006242	0.001749	0.003129	0.00187
304	0.02495	0.006785	0.01192	0.00558
448	0.07807	0.0206	0.03781	0.01749
624	0.2073	0.0555	0.09992	0.04512
832	0.4971	0.1298	0.2308	0.1052
1072	1.068	0.275	0.4795	0.2372
1344	2.314	0.5989	0.953	0.4452
1648	4.011	1.031	1.997	0.8804

#pragma In Middle of the Loop

rowlen	ijk	ijk_omp	kij	kij_omp
64	0.000152	0.000256	0.000267	0.000175
112	0.001123	0.0006	0.000689	0.00033
192	0.006755	0.001596	0.003105	0.00135
304	0.02553	0.006087	0.01217	0.004438
448	0.08001	0.02267	0.03791	0.01212
624	0.2106	0.05684	0.1004	0.03023
832	0.5143	0.1315	0.2383	0.06835
1072	1.141	0.279	0.4908	0.1422
1344	2.403	0.6119	0.9586	0.2794
1648	4.305	1.073	1.767	0.5239

#pragma Inner of the Loop

rowlen	ijk	ijk_omp	kij	kij_omp
64	0.000232	0.001363	0.000167	0.001651
112	0.00153	0.004039	0.000901	0.004354
192	0.008383	0.01233	0.004123	0.01194
304	0.02716	0.0353	0.01442	0.02872
448	0.08205	0.08467	0.04289	0.06753
624	0.2136	0.1947	0.1126	0.1351
832	0.5091	0.4198	0.2555	0.266
1072	1.089	0.8428	0.5492	0.4691
1344	2.23	1.582	1.072	0.7843

1648	4.091	2.818	1.985	1.238

Shared Loop Indices w/ Default Settings

rowlen	ijk	ijk_omp	kij	kij_omp
64	0.000155	0.000171	0.000205	0.000227
112	0.001097	0.000325	0.000727	0.000347
192	0.007524	0.001549	0.00312	0.00151
304	0.02742	0.005989	0.01214	0.005335
448	0.07866	0.02321	0.03784	0.01392
624	0.2061	0.05855	0.1001	0.0318
832	0.5018	0.1381	0.2362	0.06736
1072	1.065	0.2773	0.4822	0.1421
1344	2.261	0.6966	0.9569	0.2773
1648	3.89	1.025	1.824	0.5508

Task 2

a. Below is a table showing how OMP parallelized the baseline SOR method. After the first matrix, execution times and iterations are 10 times faster. This was done using 4 threads as NUM_THREADS for OpenMP.

size	SOR_omp time (ns)	SOR_omp iters	SOR time (ns)	SOR iters
32	1.64E+06	160	1.98E+06	102
56	6.13E+06	387	1.98E+07	376
96	2.69E+07	689	1.01E+08	651
152	1.52E+08	1578	6.09E+08	1597
224	2.63E+08	1247	1.01E+09	1244

b. Below is a table with my optimized version of MMM with the kij loop, compared to the kij loop with OpenMP. As seen on the table, performance improvements reaches to approximately 4 times if not higher. My approach was placing the loop indices into the shared list, instead of moving #pragma omp for into the middle of the loop, I added #pragma omp simd into the inner loop and vectorized the MMM method altogether.

rowlen	kij Optimized	kij OMP Default
64	4.66E-05	0.000285
112	0.000142	0.000468
192	0.000563	0.00187
304	0.001937	0.00558
448	0.005299	0.01749
624	0.01038	0.04512
832	0.021	0.1052
1072	0.0437	0.2372
1344	0.08433	0.4452
1648	0.1917	0.8804