Homework 3

Sajjad Azami - 9231031
Special Topics: Multicore Programming

May 6, 2018

Preprocessing. In this homework, we want to parallelize Merge Sort algorithm using *OpenMP*. We will consider using two methods in OpenMP, *Section* and *Task*. In order to be able to use OpenMP, we first include the library. Next, we should measure the time of the *mergeSort()* function. This can be done using *omp_get_wtime()* function. Given that we want to test our results for various number of cores, each run consists of an iteration over *num_of_threads*. For each vector, its lengths can be calculated by $s/4$ where s is the size of the vector in bytes. The rest of the report consists of the results for the two settings and a section for conclusion.

Section Based Implementation. For this part, a simple *section* structure has been implemented. The algorithm has been divided into two section, each calling *merge_section()* function recursively. Final results of the experiments are averaged over 5 runs and have been reported in Table 1.

Number of Threads	Vector Size			
	1MB	5MB	15MB	25MB
1	2.81	10.9	29.11	51.66
2	1.27	5.72	16.49	28.37
4	1.28	5.74	16.39	27.93
8	1.12	5.60	16.25	28.29

Table 1: Summary of running times in seconds for Section based implementation. Each output has been averaged over 5 runs. The results for $Number\_of\_Threads = 1$ are computed without parallelization.

For each vector size and number of threads, we then calculate the speed-up by $s_k = time_1/time_k$ where k is the number of threads. Results are summarized in Table 2.

Number of Threads	Vector Size			
	1MB	5MB	15MB	25MB
1	-	-	-	-
2	2.21	1.90	1.76	1.82
4	2.20	1.89	1.77	1.84
8	2.50	1.94	1.79	1.83

Table 2: Speed-up ratio for Section based implementation.

Task Based Implementation. For the second part, we parallelized the same function using *Task* structure. Similar to the previous section, the function has been divided into two tasks and final results have been reported in Table 3.

Number of Threads	Vector Size			
	1MB	5MB	15MB	25MB
1	0.64	3.25	9.66	15.84
2	0.35	1.82	5.33	8.90
4	0.35	1.78	5.31	8.83
8	0.36	2.00	5.47	9.03

Table 3: Summary of running times in seconds for task based implementation. Each output has been averaged over 5 runs. The results for $Number\_of\_Threads = 1$ are computed without parallelization.

Again, for each vector size and number of threads, we calculate speed-up. Results are summarized in Table 4.

Number of Threads	Vector Size			
	1MB	5MB	15MB	25MB
1	-	-	-	-
2	1.82	1.78	1.81	1.77
4	1.82	1.82	1.81	1.79
8	1.83	1.62	1.76	1.75

Table 4: Speed-up ratio for task based implementation.

Conclusion. Based on the results for both *Section* and *Task* implementations, we can observe that increasing the number of threads from 1 to 2 significantly improves the performance, almost decreasing the running time to half. Meanwhile, we can't obtain better results by adding threads, for the implementation divides the merge sort algorithm into two parts and thread numbers more than 2 are not useful.