COL730 Assignment 1
Parallel Programming | OpenMP

Shreyansh Jain
(2021MT10230)

Implementation & Design Decisions

Using Pointers

The merge_sort() and parallel_merge_sort() takes vector of Records as parameter. I created helper functions which take vector of pointers to the Records. Each pointer points to an element of the original vector. The merge function only swaps these pointers instead of moving whole Record; this *saves time by reducing number of memory actions of these large Records*.

The pointer vector is first created inside the original merge_sort() function and then is passed to the helper function which sorts it in place. This sorted pointer vector is used to finally sort the original vector. A *temp vector is used to create the sorted array* after which the original array is updated to match this temp array because changing values of the original array directly leads to errors as it causes the pointers to reference different values then intended.

Parallelizing Merge Sort

The parallel_merge_sort_helper() finds the middle index of the given portion of array and creates child tasks by recursively calling itself on the two halves of the portion. The function *does not create new divided arrays*, instead just takes the starting and ending index; this is more efficient than creating smaller copies of same array.

It waits for the tasks to complete and then merges the two halves of the portion using sequential merge().

In the base case where the range of portion of array is lesser than the threshold, *it sorts them using sequential merge sort* which is faster for small arrays.

While copying the sorted pointer vector to the original array, I use pragma omp for inside the parallel region to update the temp vector using multiple threads.

Finding the First and Last Record

In both sequential and parallel binary search, the program *finds the index of first and last occurrence of the Records with the intended key and then copies the portion in between from the sorted array*. This approach is better than the naïve approach of finding an occurrence of the key and then traversing left and right from that index until the Records have the same key; this is because the latter approach requires comparing the keys again and again for every instance. Also, my approach allows to use standard assign function which may be faster than any self-implemented copying.

In case of parallel binary search we can use an approach where if any thread finds an instance of a Record with searching key, then it appends it in the results vector, but that would require using of Synchronizations which will hinder the efficiency of the program. Also, my approach of finding the first and last index may be running in different threads in parallel thus reducing the time.

Parallelizing Binary Search

The parallel_binary_search() *divides the search range among n threads, where n is the number of threads available*. Each thread handles a portion of the array and performs binary search within that portion. The array is divided statically and equally between the threads because the chance of finding the key is equal in all portions and dynamically dividing them will not affect that but only add the scheduling overhead.

I tried implementing another approach in which each thread recursively divides the array and creates a task for searching in that portion, and this may work well in theory but is performed worst in practice as the overhead of creating these tasks is greater than simply comparing the values with the key.

In my approach, instead of using parallel for I am using thread_id to divide the portion to each thread. *Same thread is assigned same portion for finding both first and last occurrence of the key*. This may seem counterintuitive as there is a higher chance that the first and last key is in same portion and thus only one thread is doing the searching while we could have used two threads for it. But the performance proved otherwise. This could be because the same thread has all the values of the portion already loaded in the Cache and thus the overhead of updating the cache for loading values from other portion is not encountered in my approach which proves better than parallelizing two binary searches.

The parallel version of binary search works slower than the sequential version, again because of the overhead. In theory dividing the search to k threads may change the time complexity to O(log(n/k)) but in practice creating k threads and assigning each portion to them is O(k) task which proves slower than the simple approach of comparing k values of the array and entering the portion which may contain the key.

Challenges faced

Determining the Scope of Variable

I was getting wrong output in parallel_merge_sort() and spent a lot of time debugging. Also used default(none) in the pragma omp parallel inside parallel_merge_sort() to know if any variable was not correctly scoped. That didn't solved the issue but at last I tried using *shared clause while creating tasks* and found that it was the cause of wrong output. Faced similar problems throughout the assignment.

Slowdown in Binary search

While running binary search for multiple keys on same array, the time taken was much more than n*Time for one binary search; where n is no. of keys to search. Spent a lot of time finding the cause and finally found that *while searching for the last occurrence of any key, I was giving left(the index of first occurrence of key) as the start parameter* (which actually makes sense to reduce the range of searching). But it turns out that was slowing down when executing for multiple keys iteratively because of changing the cache again and again while the effect of reducing the range of binary search is not really significant. Thus using 0 only as the start proved to perform better over multiple keys.

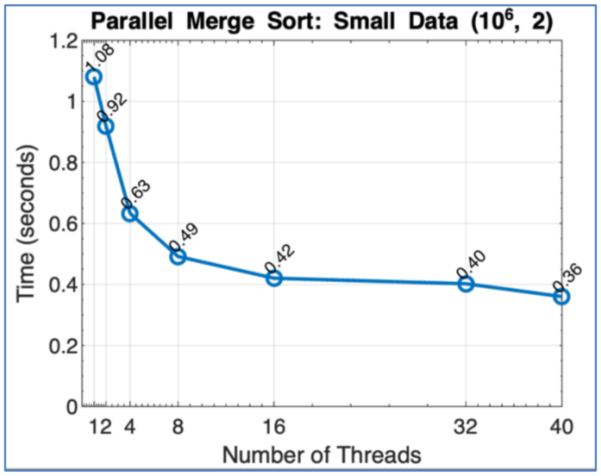
Performance Analysis

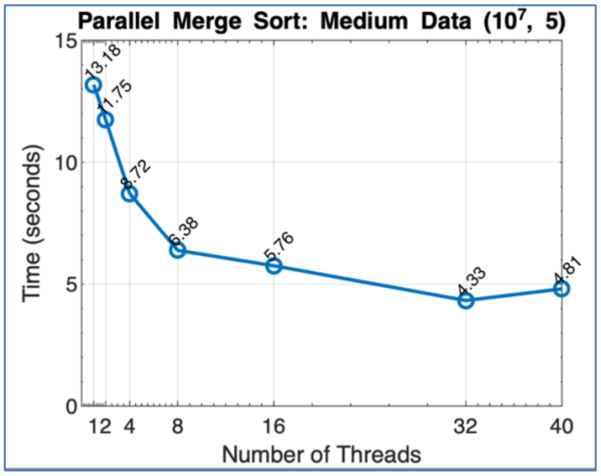
Parallel Merge Sort

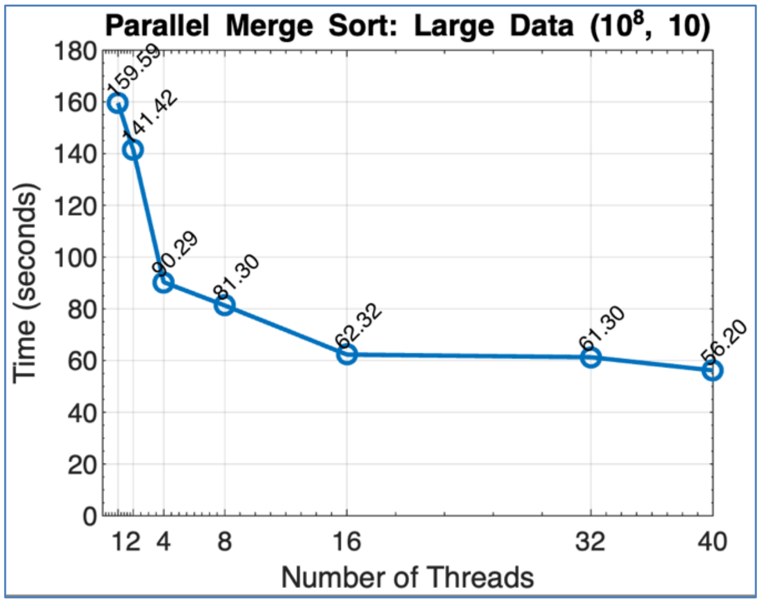
Here are the results for Sequential and Parallel Merge Sort:

No. of Threads	Time Taken (10^6)	Speedup	Time Taken (10^7)	Speedup	Time Taken (10^8)	Speedup	Efficiency
1	1.080136	1	13.180083	1	159.586704	1	100%
2	0.918646	1.176	11.752350	1.121	141.420158	1.129	58.82%
4	0.633028	1.706	8.719071	1.511	90.291773	1.768	42.95%
8	0.491618	2.197	6.384690	2.064	81.299355	1.964	24.55%
16	0.420473	2.569	5.755063	2.291	62.322155	2.561	16.00%
32	0.402272	2.684	4.330394	3.044	61.298427	2.604	8.39%
40	0.360195	3.000	4.807634	2.742	56.198805	2.839	7.50%

Plots for Time Taken(seconds) vs Number of Threads in Parallel Merge Sort:





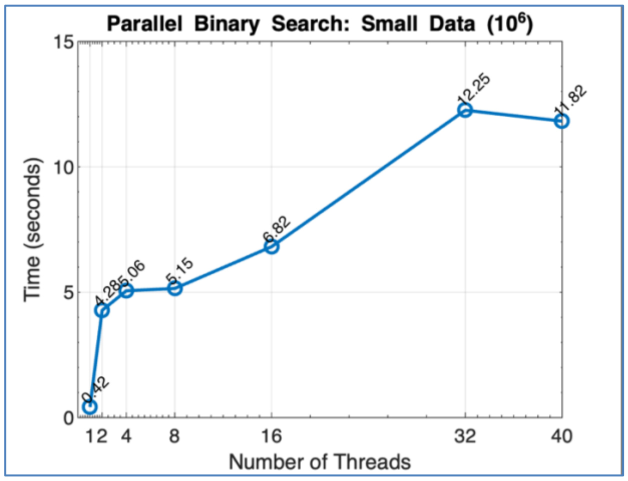


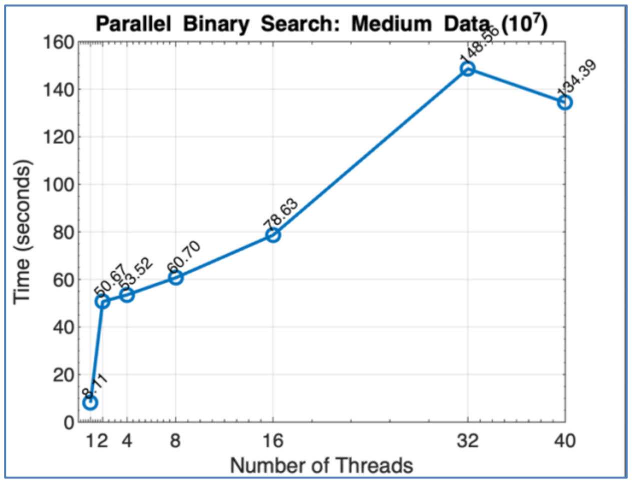
Parallel Binary Search

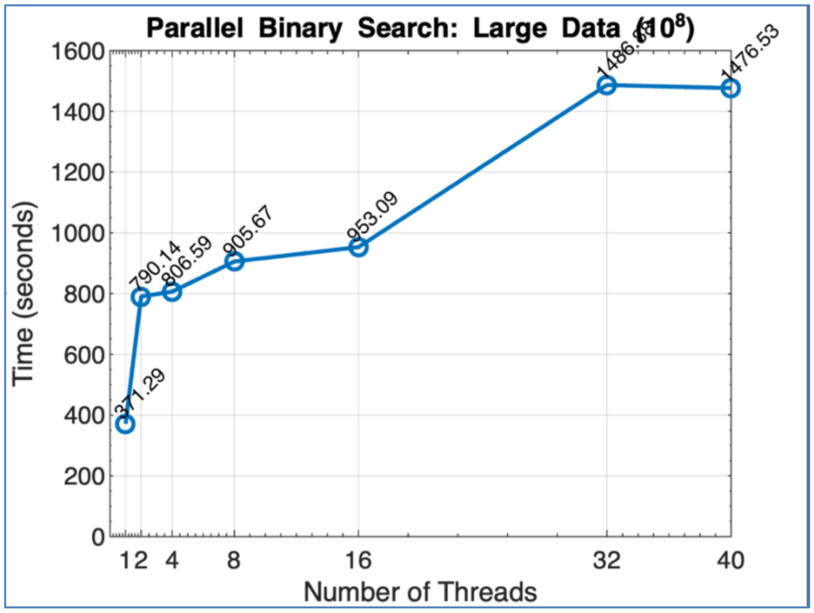
Here are the results for Sequential and Parallel Binary Search:

No. of Threads	Time Taken (10^6)	Speedup	Time Taken (10^7)	Speedup	No. of Comparisons	Time Taken (10^8)	Sp.	Eff.
1	0.421526	1.00	8.106169	1.00	5.7745737	371.292330	1.00	100%
2	4.277361	0.098	50.668417	0.16	8.4739405	790.142037	0.47	23.53%
4	5.059004	0.083	53.516928	0.15	12.2735127	806.591708	0.46	11.76%
8	5.148429	0.082	60.698061	0.13	20.0731048	905.668717	0.41	5.13%
16	6.815916	0.062	78.625683	0.10	35.8726277	953.088348	0.39	2.44%
32	12.254177	0.034	148.555204	0.05	67.6721493	1486.880794	0.25	0.78%
40	11.820103	0.036	134.394338	0.06	83.5978858	1476.533627	0.25	0.63%

Plots for Time Taken(seconds) vs Number of Threads in Parallel Binary Search:







Cache Performance Analysis

Here are the results for sequential and parallel merge sort:

No. of Threads	**Cache References**	**Cache Misses**	**Percentage**
1	23,98,92,297	10,86,33,103	45.284 %
2	37,18,29,184	17,82,62,364	47.942 %
4	39,13,08,949	15,42,60,573	39.422 %
8	49,04,98,288	20,06,61,152	40.910 %
16	50,36,48,276	23,48,81,464	46.636 %
32	63,05,49,292	23,37,27,346	37.067 %
40	71,51,22,794	22,02,80,103	30.803 %

The percentage of cache misses is *generally decreasing as the number of threads increase*. As the number of threads increases, each thread works on a smaller portion of the data. This can lead to better utilization of cache lines, as each thread's working set becomes smaller and fits better into its local cache. There is also an opposing factor of increasing false sharing and cache contention as number of threads increase. This might be the reason for increasing cache misses from sequential to 2 threads and then increase in cache misses from 8 to 16 threads.

The Cache References and Misses of Sequential and Parallel binary search were *similar for all number of threads*. The parallelization of binary search is done by dividing the search space among threads and having each thread perform independent searches. This doesn't change the fundamental memory access pattern of binary search.