The algorithm in sequential sort is radix sort. And the algorithm in parallel sort is quick sort.
The main factors concerned when choosing these two algorithms are sorting time and storage space. For sequential sorting, the main factor concerned is sorting time due to the relative small numbers. Because the boundary of comparison algorithms which is O(nlgn), the non-comparison algorithm shows an advantage. So counting sort, bucket sort, radix sort, sample sort are the four algorithms taken into consideration. The algorithm chosen at last is radix sort, the reasons are below:
1.Radix sort needs smaller store space compared with counting sort. Besides it can sort all kinds of numbers(counting sort can only sort discrete value, bucket sort needs a good distribution of numbers to perform well).
2.Radix sort is fast,the sorting time of which is O((n+b) * logb(k))(b is number base, k is maximum value). Because of the limited digits in input.txt, which makes radix sort perform well.
3.Radix sort is simple, clear and clever.
4.I am trying to combine radix sort with sample sort in parallel sorting.

For parallel sorting, the first algorithm I am trying to use is sample sort. Because sample sort shows a big advantage in sorting big numbers. There are several steps to implement:

1. Divide a given array into k parts equally.
2. Sort each part.
3. pick out k pivots in each part by certain stride.
4. sort k^2 pivots and pick out k pivots by certain stride.
5. use the k final pivots to form k buckets.
6. sort k buckets.

The main advantage of bucket sort is the low time cost when there is a good number distribution. Sample sort is trying to find good "bounds" to distribute the number properly so as to implement bucket sort and the cost of this improvement is not heavy because the additional steps are mostly implemented in pivots. The time cost is O(n/p+log(p))+O(p)+O(log(p))+O((n/p)log(p))+O(c/p).(p is the pivots selected, c is the sequential sorting algorithm underlying). So the approximate time cost of sample sort is O(n)+O(c).

At first I am trying to use radix sort in it. However I find that radix sort requires too much space. And with the digits grow, there are less advantages of radix sort compares with quick sort which needs much less store space. So finally, quick sort and sample sort are combined to implement parallel sorting. However when I finish the code of sample sort, there is a big problem with debugging(core dump). Because of the large amount of code, time constrain and being a beginner in openmp.I feel it is impossible to finish the assignment in a short time. So I am trying to find a practical algorithm to implement, in the mean time I realize that the code structure of quick sort (recursion) is a huge advantage to implement parallel sorting. In quick sort the algorithm is to recursively partition an array to two parts, one of which is smaller than pivot and the other one is bigger than pivot. This means every partition is an independent work and it is possible to implement them in parallel. So I use (pragma omp task) to implement recursively. An the result of real time is among 0.431-0.474 for 1 million numbers to sort. And the 131 code lines is much less than which in the sample is 272.
At last, parallel quick sort has a big disadvantage which is the extreme slow to sort already sorted numbers, so I add one more parallel merge sort and hope to make it better.

	Inp.txt			Average	Inp-large.txt			Average
Seq_Sort	7.512	7.535	7.629	7.558	767.133	764.455	772.356	767.981
Par_Sort	2.492	2.622	2.464	2.540	235.721	233.684	240.543	236.649
Sequential radix sort	0.156	0.154	0.158	0.156	0.475	0.478	0.456	0.470
Parallel quick sort	0.082	0.083	0.082	0.082	0.474	0.431	0.458	0.454
Parallel merge sort	0.162	0.108	0.106	0.125	0.843	0.846	0.866	0.852