Sorting using OpenMP

ASSIGNMENT 2 COL 730
2018MT60778
ASHWINI KUMAR

In this assignment we have implemented parallel versions of quick sort, merge sort, and radix sort using OpenMP. The input was provided as pointer of array of pSort::dataType to the void sort function in the pSort library. The keys of each data was a signed integer upto 32 bits. Each of the sort algorithms have been described below:

MERGE SORT:
In this sort, we divide the input array of the present call into left and right halves. We create 2 threads in the present call and then assign the first thread created to make a recursive call on the left half of the input and assign the other thread to make a recursive call on the right half of the input. If the input size is 2 then compare the two elements and then swap them if the second element is smaller than the first element. If size is 1 then end the recursive call.
This way left and right half compute in parallel in the two threads.

QUICK SORT:
In this sort, we similarly divide the input array of the present call into left and right partitions after calculating the partition index. Partition index is the index at which the input array is divided into left and right subarrays. Create 2 sections and then assign the first section to make a recursive call on the left subarray(similar to merge sort algorithm) and assign the second section to make a recursive call on the right subarray. If size of input array is less than 2^18, then call the sequential version of quick sort.

RADIX SORT:
We start to perform radix sort from LSD(Least Significant DIgit) and taking base 10.In this sort, we create a 3-d matrix in the shared memory for all the threads such that first dimension represents the digit from 0 to 9 in which we will later keep the local buckets obtained from each thread. Second dimension represents the number of threads and the third dimension has the local bucket that will be created in each thread. We set the total number of threads to be equal to the number of processors available.
Then each thread in parallel get their portion of shared memory and are allowed to make 10 local buckets in which they can sort using bucket sort. These local buckets are pushed into the 3-D matrix in the shared memory according to the thread number. So, for the present digit we then collect all the buckets to form a new data array and then pass it into the next iteration of the digit moving left from LSD.
This way after 15 iterations on each digit moving left from LSD, we get the sorted input array. Since we have signed numbers as keys then we perform another operation similar to the above operation. We create two buckets this time instead of 10 buckets and then perform the exact same operations to create threads equal to the number of processors. Then we again create local buckets for each thread and then collect them again to create a new data array. This time

while we collect the negative numbered keys data, we store them in the opposite order. While collecting the positive numbered keys data, we store them in the forward order only.
So, the given input array having signed keys is now sorted.

MAJOR DECISIONS:
The major decisions made were that:
(1) Sequential algorithms are much more time taking in comparison to parallel algorithms. Still all parallel algorithms use some kind of serialization because it was not possible to sort parallely when there is only one processor available such as in the case of quick sort. Parallel algorithms proved to be much faster as shown in the tables below.
(2) In all the sorting functions, if it is not returning a pointer array then we have to create a temporary pointer pointing at the same location as that of input data and then copy all the sorted data into the temporary pointer array. If we do not do so, then there will be no change in the data array when the function returns.
(3) In the merge function used for merge sort, a duplicate array was passed. Since, it was giving a segmentation fault when we do not pass a duplicate array. It was previously creating two arrays of large size which was taking up extra memory. When a duplicate array was passed instead of creating two new arrays, it was workingproperly and with good efficiency.
(4) In Quick sort, we sort sequentially for the data when input size is less than 2^18. If we do not do so, then it creates more threads for such a short input size which lead to an increase in time taken by the program.
(5) In merge function used for merge sort, I had to use the #pragma omp for instead of conditioning on the thread number as we cannot determine what will be the thread numbers of the two created threads in a call for merge sort function. Using #pragma omp for gave correct results while using the condition of checking the thread number gave incorrect results in many cases.

EXPERIMENTS:
Here is a slight comparison of all the three sorting algorithms using 12 logical processors and various input sizes(time is taken in seconds) on my laptop:

QUICK SORT:

SIZE OF INPUT	Time Taken
2^10	0.000318627
2^15	0.003530493
2^18	0.0383349
2^20	0.118931
2^22	0.39033

2^24	1.92097
2^26	9.01595

MERGE SORT:

SIZE OF INPUT	Time Taken
2^10	0.00168208
2^15	0.0098825
2^18	0.0837457
2^20	0.343039
2^22	1.42095
2^24	5.95737
2^26	28.596

RADIX SORT:

SIZE OF INPUT	Time Taken
2^10	0.00312351
2^15	0.0173814
2^18	0.110034
2^20	0.417004
2^22	1.52871
2^24	5.94514
2^26	23.5119

The above tables are for the purpose of comparison only. The functions were also run on HPC(High Performance Computing) IITD Servers and showed similar results.

BEST SORTING: Quick sort proved to be the best sorting algorithm among all three sorting algorithms which can be observed from the above tables(take a fixed input size and then on comparing quick sort has the least time taken).

SCALABILITY:
As demonstrated above, we see that:

Quick Sort: It scales efficiently from a very small input size to 2^26. This was experimented on

HPC IITD Server as well. It scaled up to 2^30 input size and 20 processors. Quick sort showed the best performance among the 3 sorting functions at that size also.
Merge Sort: It scales efficiently from a very small input size to 2^26. This was experimented on HPC IITD Server as well. It scaled up to 2^30 input size and 20 processors.
Radix Sort: It scales efficiently from a very small input size to 2^26. This was experimented on HPC IITD Server as well. It scaled up to 2^30 input size and 20 processors.