PD_10_Kushagra Suryawanshi

HPC LAB ASSIGNMENT 5

Title:
Write a C Program for parallel Quick Sort or Breadth First Search or Depth first Search of a large array of integers using OpenMP.

Aim: To compare the execution performance of a C Program for parallel Quick Sort or Breadth First Search or Depth first Search with respect to its serial approach for a large array of integers using OpenMP.

Objective:
1. Write sequential Quick Sort or Breadth First Search or Depth first Search
2. Calculate complexity of the program
3. Write parallel Quick Sort or Breadth First Search or Depth first Search
4. Calculate complexity of the program
5. Measure and compare time taken by the parallel Quick Sort or Breadth First Search or Depth first Search with respect to the serial approach

Theory:

A}Write about the serial Quick Sort or Breadth First Search or Depth first Search
(choose any one algorithm)
->QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
 if (low < high)
 {
 /* pi is partitioning index, arr[pi] is now
 at right place */
 pi = partition(arr, low, high);
 quickSort(arr, low, pi - 1); // Before pi
 quickSort(arr, pi + 1, high); // After pi
 }
```

}

B}Write about the parallel Quick Sort or Breadth First Search or Depth first Search (choose any one algorithm)
->We consider the case of distributed memory:
Each process holds a segment of the unsorted list.The unsorted list is evenly distributed among the processes.
Desired result of a parallel quicksort algorithm:

1. The list segment stored on each process is sorted
2. The last element on process i's list is smaller than the first element on process i + 1's list
3. We randomly choose a pivot from one of the processes and broadcast it to every process
4. Each process divides its unsorted list into two lists: those smaller than (or equal) the pivot, those greater than the pivot.
5. Each process in the upper half of the process list sends its "low list" to a partner process in the lower half of the process list and receives a "high list" in return.
6. Now, the upper-half processes have only values greater than the pivot, and the lower-half processes have only values smaller than the pivot.
7. Thereafter, the processes divide themselves into two groups and the algorithm recurses.
8. After log P recursions, every process has an unsorted list of values completely disjoint from the values held by the other processes.
9. The largest value on process i will be smaller than the smallest value held by process i + 1.
10. Each process can sort its list using sequential quicksort. This parallel quicksort algorithm is likely to do a poor job of load Balancing.
11. If the pivot value is not the median value, we will not divide the list into two equal sublists. Finding the median value is prohibitively expensive on a parallel computer. The remedy is to choose the pivot value close to the true median!

C} Justify the usage of OpenMP in the parallel Quick Sort or Breadth First Search or Depth first Search.
a) If we use #pragma omp parallel section inside quicksort function the total no of threads will be equal to the total no of calls to the quicksort function. The code is correct, but two threads are created and distroyed in each call which creates a lot of overhead which makes it slower than the serial code.
b) A few things: we could have used the sections construct with similar code. This was generally the way to do things long ago, and the more modern and preferred alternative is to use tasks. In my experience, tasks also perform better, because with sections, each thread only does the work for one section (which is equivalent to the tasks above), whereas each thread can perform the work of many tasks

Test on data set of sufficiently large size.
Compute Total cost and Efficiency as :-

1.Total Cost = Time complexity × Number of processors used

2.Speedup=Ts/Tp [Ts : Serial execution time / Parallel execution time]

3.Speedup = (Execution time of sequential algorithm/Execution time of the parallel algorithm)

4.Efficiency = Speedup/p
Where p : number of processors
Mention the number of processors / processor cores of your machine.
Consider data points as follows :-
Auto Generate the data points/values through :-

Sr.No	Data Points/Data Values	Time taken for Serial Program	Time Taken for Parallel Program
1.	500	0.000056	0.013058
2.	1000	0.000112	0.030379
3.	10000	0.001432	0.291900
4.	50000	0.391000	1.882000
5.	60000	0.467000	0.765400
6.	1000000	122.747000	110.376000

a) Giving a specific pattern of data value generation
b) Randomizing the data points/values
(Also a file may be maintained for reading the values stored)

<u>Input</u>: Unsorted array of data points/values.
<u>Output</u>: Sorted Array of data points/values.
<u>Platform:</u> Ubuntu (give latest version) or Windows 8.

<u>Conclusion</u>: Thus, successfully studied, analysed serial and parallel Quick Sort or Breadth First Search or Depth first Search.

<u>FAQs:</u>

Q. Explain important properties of ideal sorting algorithm and complexity analysis of quick sort.

1. The quick sort is an in-place, divide-and-conquer, massively recursive sort algorithm.
2. The efficiency of the algorithm is majorly impacted by which element is chosen as the pivot point.
3. The worst-case efficiency of the quick sort is o(n^2) when the list is sorted and left most element is chosen as the pivot. As long as the pivot point is chosen randomly, the quick sort has an algorithmic complexity of O(nlog(n)).
4. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge,If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort Stable i.e. equal keys aren't reordered.
5. Operates in place, requiring O(1) extra space.
6. Worst-case O(n·lg(n)) key comparisons.
7. Worst-case O(n) swaps.

8. Adaptive: Speeds up to O(n) when data is nearly sorted or when there are few unique keys.

Analysis of QuickSort
Time taken by QuickSort, in general, can be written as following.
T(n) = T(k) + T(n-k-1) + (n)
The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.
A]Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order.
T(n) = T(n-1) + Q(n)
The solution of above recurrence is O(n2).
B] Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.
T(n) = 2T(n/2) + Q(n)
The solution of above recurrence is O(nLogn)
C]Average Case:
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.
T(n) = T(n/9) + T(9n/10) + (n)
Solution of above recurrence is also O(nLogn)

9. Explain steps required for parallel algorithm development.
Ans: The process of designing a parallel algorithm consists of four steps:
1.Decomposition of a computational problem into tasks that can be executed simultaneously, and development of sequential algorithms for individual tasks
2.Analysis of computation granularity
3.Minimizing the cost of the parallel algorithm
4.Assigning tasks to processors executing the parallel algorithm.

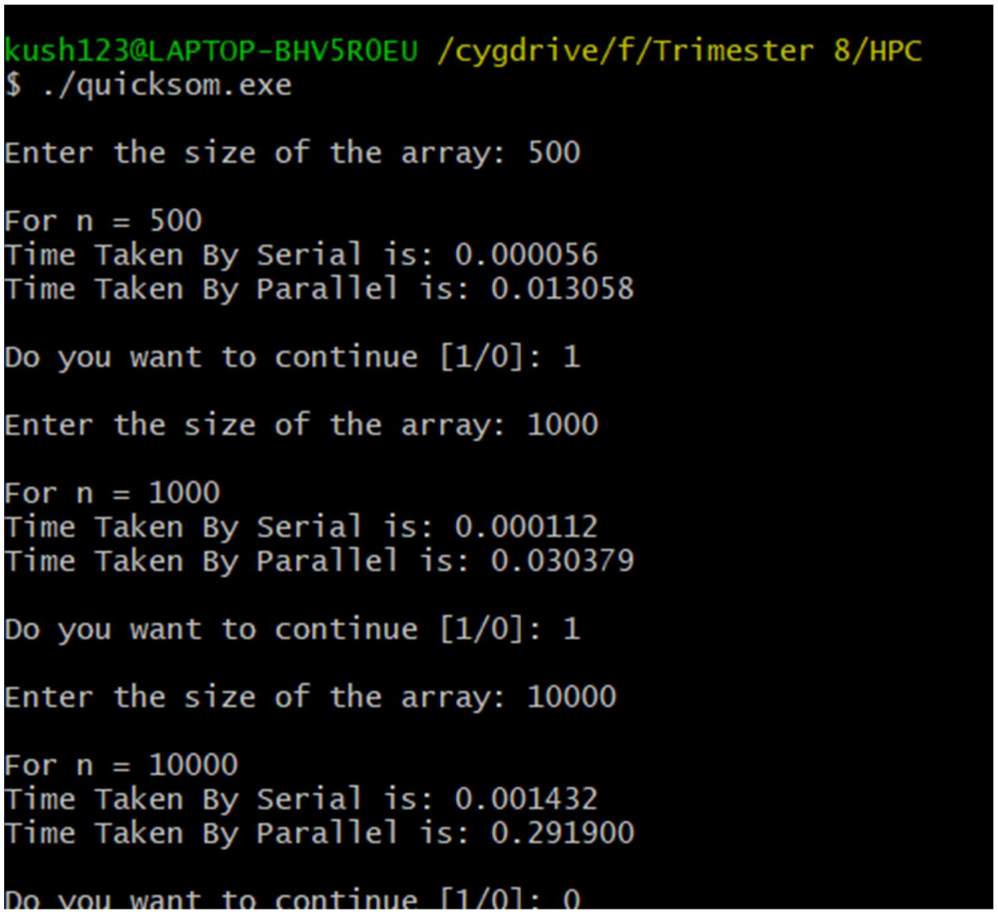
Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define NUM_THREAD 4;
int Partition(int l, int h, int arr[])
{
int pivot = arr[h];
int i = l - 1;
int temp;
for (int j = l; j <= h - 1; j++)
```

```
{
if (arr[j] < pivot)
{
i++;
temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
}
}
temp = arr[i + 1];
arr[i + 1] = arr[h];
arr[h] = temp;
return (i + 1);
}
void Q_Sort_serial(int l, int h, int arr[])
{
if (l < h)
{
int p = Partition(l, h, arr);
Q_Sort_serial(l, p - 1, arr);
Q_Sort_serial(p + 1, h, arr);
}
}
void Q_Sort_parallel(int l, int h, int arr[])
{
if (l < h)
{
int p = Partition(l, h, arr);
#pragma omp parallel sections
{
#pragma omp section
{
Q_Sort_parallel(l, p - 1, arr);
}
#pragma omp section
{
Q_Sort_parallel(p + 1, h, arr);
}
}
}
}
int main()
{
long long int i, n;
int ch = 1;
```

```
while (ch == 1)
{
printf("\nEnter the size of the array: ");
scanf("%lld", &n);
int *arr, *arr1;
arr = (int *)malloc(sizeof(int) * n);
arr1 = (int *)malloc(sizeof(int) * n);
printf("\nFor n = %lld\n", n);
for (i = 0; i < n; i++)
{
arr[i] = rand() % 1000;
arr1[i] = arr[i];
}
int total_threads = NUM_THREAD;
omp_set_num_threads(total_threads);
double t_start = omp_get_wtime();
Q_Sort_serial(0, n - 1, arr);
double t_end = omp_get_wtime();
printf("Time Taken By Serial is: %lf\n", t_end - t_start);
t_start = omp_get_wtime();
Q_Sort_parallel(0, n - 1, arr1);
t_end = omp_get_wtime();
printf("Time Taken By Parallel is: %lf\n", t_end - t_start);
free(arr);
free(arr1);
printf("\nDo you want to continue [1/0]: ");
scanf("%d", &ch);
}
return 0;
}
```

OUTPUT:



```
kush123@LAPTOP-BHV5R0EU /cygdrive/f/Trimester 8/HPC
$ ./quicksom.exe

Enter the size of the array: 500

For n = 500
Time Taken By Serial is: 0.000056
Time Taken By Parallel is: 0.013058

Do you want to continue [1/0]: 1

Enter the size of the array: 1000

For n = 1000
Time Taken By Serial is: 0.000112
Time Taken By Parallel is: 0.030379

Do you want to continue [1/0]: 1

Enter the size of the array: 10000

For n = 10000
Time Taken By Serial is: 0.001432
Time Taken By Parallel is: 0.291900

Do you want to continue [1/0]: 0
```