Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

PP LAB-3

Name: Raghav Maheshwari

Roll: PA53

Panel: A

* Problem statement:

Write a parallel program for quicksort algorithm execute it on multiprocessor system using openMp section directives.

* Aim:

To write a C program to sort a large vector using quick sort.

* Objectives:
To understand need of parallelism and nested parallel threads.

* Theory:

- Serial quick sort procedure

i) Select one of nos. as pivot.
ii) Divide arrays into 2 subarrays a low array contains nos. smaller than pivot and a high array contains nos. higher than pivot.

iii) The low array and high array repeat the process recursively to sort them.

iv) The final sorted resultant is concatenate of sorted low array and pivot & sorted high array.

* Parallel quick sort:

i) In this we consider case of distributed memory

ii) Each process held a segment of the insorted list

iii) The unsorted list is evenly distributed among the process.

iv) Desired result of parallel quick sort.

a) The array segment stored on each process is sorted array.

b) The last element on process is small than first element on process its array.

- usage of OpenMp in parallel quicksat

In this we divide it into two parts by partition given array arround its pivot where each part is a process by independent thread i.e. diff. threads will find out pivot element

recursively and sort array independent. If we use pragma omp the total no. of threads will be equal to total no. of calls to the qsuick sort function. The code is correct but 2 threads are created and destroyed in each cell which creats lot of overlooded which make it slower.

=> Total cost = Time complexity x no. of processes used

$= O(n \log n) \times 1$

→ Speed up $= \frac{T_s}{T_p} = \frac{0.050669}{4.560751}$

$= 0.011109$

=> Efficiency $= \frac{\text{speed up}}{p}$

$= \frac{0.011109}{4}$

$= 0.0027$

$= 0.27\%$.

where p is no. of processors or our machine.

recursively and sort array independent. If we use pragma omp the total no. of threads will be equal to total no. of calls to the qsuick sort function. The code is correct but 2 threads are created and destroyed in each cell which creat lot of overlooded which make it slower.

=) Total cost = Time complexity x no. of processes used

$= O(n \log n) \times 4$

→ Speed up = $\frac{T_s}{T_p} = \frac{0.050669}{4.560754}$

$= 0.011109$

=) Efficiency = $\frac{\text{speed up}}{p}$

$= \frac{0.011109}{4}$

$= 0.0027$

$= 0.27\%$.

where p is no. of processors or our machine.

recursively and sort array independent. If we use pragma omp the total no. of threads will be equal to total no. of calls to the qsuick sort function. The code is correct but 2 threads are created and destroyed in each cell which creat lot of overloaded which make it slower.

=> Total cost = Time complexity x no. of processes used
= O(n log n) x 1

→ Speed up = $\frac{Ts}{Tp} = \frac{0.050669}{4.560751}$

= 0.011109

=> Efficiency = $\frac{\text{speed up}}{p}$

= $\frac{0.011109}{4}$

= 0.0027

= 0.27%.

where p is no. of processors or our machine.

Input:

Unsorted array of data points/values.

Output:

Sorted Array of data points/values.

Platform:

Ubuntu (v20.04)

Conclusion:

In this practical we learnt how to make and use parallel quick sort algorithm using openMP and its usage over sequential quick sort algorithm.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

long long int partition(double *a, long long int low, long long int high)
{
   double pivot = a[low];
   long long int i = low + 1;
   long long int j = high;
   do
   {
      while (a[i] <= pivot)
      {
         i++;
      }
      while (a[j] > pivot)
      {
         j--;
      }
      if (i < j)
      {
         double temp = a[i];
         a[i] = a[j];
         a[j] = temp;
      }
   } while (i < j);
   double temp = a[low];
   a[low] = a[j];
   a[j] = temp;
   return j;
}
long long int partitionParallel(double *a, long long int low, long long int high)
{
   double pivot = a[low];
   long long int i = low + 1;
   long long int j = high;
#pragma omp parallel
   do
   {
      while (a[i] <= pivot)
      {
         i++;
      }
      while (a[j] > pivot)
      {
         j--;
```

```
        }
        if (i < j)
        {
            double temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    } while (i < j);
    double temp = a[low];
    a[low] = a[j];
    a[j] = temp;
    return j;
}

void quickSort(double *a,long long int low,long long int high)
{
    long long int partitionIndex;
    if (low < high)
    {
        partitionIndex = partition(a, low, high);
        quickSort(a, low, partitionIndex - 1);
        quickSort(a, partitionIndex + 1, high);
    }
}

void quickSortParallel(double *a, long long int low, long long int high)
{
    long long int partitionIndex;
    if (low < high)
    {
        partitionIndex = partitionParallel(a, low, high);
#pragma omp parallel sections shared(a)
        {
#pragma omp section
            quickSortParallel(a, low, partitionIndex - 1);
#pragma omp section
            quickSortParallel(a, partitionIndex + 1, high);
        }
    }
}

int main()
{
    double *arr;
    long long int n;
    printf("\nEnter the size of the array: ");
    scanf("%lld", &n);
```

```
    arr = (double *)malloc(n * sizeof(double));
    for (long long int i = 0; i < n; i++)
    {
        arr[i] = rand() % (i + 1);
    }
    double t_start, t_start1, t_end, t_end1, t_diff, t_diff1;
    t_start = omp_get_wtime();
    quickSort(arr, 0, n - 1);
    t_end = omp_get_wtime();
    t_diff = t_end - t_start;
    printf("The time taken by serial QuickSort algo is: %f", t_diff);
    printf("\n");
    t_start1 = omp_get_wtime();
    quickSortParallel(arr, 0, n - 1);
    t_end1 = omp_get_wtime();
    t_diff1 = t_end1 - t_start1;
    printf("The time taken by parallel QuickSort algo is: %f", t_diff1);
}
```

Input and Output:

Size of array	Ts	Tp
500	0.00009	0.004199
1000	0.004199	0.007958
5000	0.001151	0.027703
10000	0.002479	0.074203
15000	0.003883	0.142914
20000	0.005346	0.231012
30000	0.006813	0.476644
50000	0.011106	1.253187
70000	0.015602	2.342934