FALL – SEMESTER **Name: Nidhi Singh**
Course Code: MCSE503P **Reg. No:22MAI0015**
Course-Title: – Computer Architecture and Organization
DIGITAL ASSIGNMENT - V
(LAB) **Slot-** L53+L54

---

1. Write an Open MP program using C for the following:

a) **Quick sort**

b) **Minimum spanning tree**

Quick sort :-

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

int k = 0;

int partition(int arr[], int low_index, int high_index)
{
   int i, j, temp, key;
   key = arr[low_index];
   i = low_index + 1;
   j = high_index;
   while (1)
   {
      while (i < high_index && key >= arr[i])
         i++;
      while (key < arr[j])
         j--;
      if (i < j)
      {
         temp = arr[i];
         arr[i] = arr[j];
         arr[j] = temp;
      }
      else
      {
         temp = arr[low_index];
         arr[low_index] = arr[j];
         arr[j] = temp;
         return (j);
      }
   }
}
```

```
void quicksort(int arr[], int low_index, int high_index)
{
    int j;
    if (low_index < high_index)
    {
        j = partition(arr, low_index, high_index);
        printf("Pivot element with index %d has been found out by thread %d\n", j, k);

#pragma omp parallel sections
        {
#pragma omp section
            {
                k = k + 1;
                quicksort(arr, low_index, j - 1);
            }

#pragma omp section
            {
                k = k + 1;
                quicksort(arr, j + 1, high_index);
            }
        }
    }
}

int main()
{
    int arr[100];
    int n, i;

    printf("Enter the value of n\n");
    scanf("%d", &n);
    printf("Enter the %d number of elements \n", n);

    for (i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }

    quicksort(arr, 0, n - 1);

    printf("Elements of array after sorting \n");

    for (i = 0; i < n; i++)
    {
        printf("%d\t", arr[i]);
    }

    printf("\n");
}
```

Output :-

```
Enter the value of n
5
Enter the 5 number of elements
40
20
43
10
96
Pivot element with index 2 has been found out by thread 0
Pivot element with index 0 has been found out by thread 1
Pivot element with index 3 has been found out by thread 2
Elements of array after sorting
10      20      40      43      96
```

Minimum spanning tree :-

```
#include<stdio.h>
#include<stdlib.h>
int i, j, k, a, b, u, v, n, ne = 1;
int min, mincost = 0, cost[100][100], parent[100];
int find(int);
int uni(int, int);
void main()
{
   printf("\n\n\tImplementation of Kruskal's algorithm\n\n");
   printf("\nEnter the no. of vertices\n");
   scanf("%d", &n);
   printf("\nEnter the cost adjacency matrix\n");
   for (i = 1; i <= n; i++)
   {
      for (j = 1; j <= n; j++)
      {
         scanf("%d", &cost[i][j]);
         if (cost[i][j] == 0)
            cost[i][j] = 999;
      }
   }
   printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
   while (ne < n)
   {
      for (i = 1, min = 999; i <= n; i++)
      {
         for (j = 1; j <= n; j++)
         {
            if (cost[i][j] < min)
            {
```

```
                min = cost[i][j];
                a = u = i;
                b = v = j;
            }
        }
    }
    u = find(u);
    v = find(v);
    if (uni(u, v))
    {
        printf("\n%d edge (%d,%d) =%d\n", ne++, a, b, min);
        mincost += min;
    }
    cost[a][b] = cost[b][a] = 999;
  }
  printf("\n\tMinimum cost = %d\n", mincost);
}
int find(int i)
{
  while (parent[i])
    i = parent[i];
  return i;
}
int uni(int i, int j)
{
  if (i != j)
  {
    parent[j] = i;
    return 1;
  }
  return 0;
}
```

Output :-

```
        Implementation of Kruskal's algorithm

Enter the no. of vertices
5

Enter the cost adjacency matrix
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 1 0 9
1 5 7 9 0

The edges of Minimum Cost Spanning Tree are

1 edge (4,3) =1

2 edge (5,1) =1

3 edge (1,2) =2

4 edge (2,3) =3

        Minimum cost = 7
```

Question 2

Compare and contrast open MP with MPI. Write a c program using MPI to perform the following:

a) Arithmetic operations

b) To simulate send and receive message Communication

OpenMP	MPI
High-level API allowing shared-memory parallel computing	High-level implementation of Message Passing Interface (MPI) for distributed-memory systems.
Allows parallel code to run on a single multi-core system	Allows parallel code to run on multiple systems connected by a network
Automatically creates multiple threads and deals with synchronization	Provides API that allows programmer to control communication between distributed nodes
Automatically reduces/compiles the final results	Programmer has to manually receive and compile the results
Can run offline in isolation	Needs a network to function

Arithmetic Operations

```
#include <mpi.h>
#include <stdio.h>

void main()
{
   int rank, size, a, b, sum, sub, mul, div, rem;
   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (rank == 0)
   {
      printf("Enter the value of a and b\n");
      scanf("%d%d", &a, &b);
      MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
      MPI_Send(&b, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
   }
   else
   {
      MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Recv(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      sum = a + b;
      sub = a - b;
      mul = a * b;
      div = a / b;
      rem = a % b;
      printf("The Sum of a and b is %d", sum);
```

```
        printf("The difference of a and b is %d", sub);
        printf("The Product of a and b is %d", mul);
        printf("The Quotient and Remainder of a and b is %d and %d", div, rem);
    }
    MPI_Finalize();
}
```

Output :-

```
Enter the value of a and b
5 10
The Sum of a and b is 15
The difference of a and b is -5
The Product of a and b is 50
The Quotient and Remainder of a and b is 0 and 5
```

Simulate send and receive message Communication

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    int processRank, sizeOfCluster, messageItem;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &processRank);
    MPI_Comm_size(MPI_COMM_WORLD, &sizeOfCluster);
    if (processRank == 0)
    {
        messageItem = 42;
        MPI_Send(&messageItem, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Message Sent: %d\n", messageItem);
    }
    else if (processRank == 1)
    {
        MPI_Recv(&messageItem, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Message Received: %d\n", messageItem);
    }
    MPI_Finalize();
    return 0;
}
```

Question 3:-

Study about VTune performance analyser tool to profile the program code.

Intel VTune Performance Analyzer is a commercial application for software performance analysis which detects performance bottlenecks (hotspots) in an application and performs optimization for systems running on Intel processors.

It is the software development rate which is brought to the forefront nowadays. However, we should not forget about optimization which, despite the common belief, has always been highly important too. Intel VTune Performance Analyzer is one among a number of tools designed to assist developers in software optimization. The tool can detect and locate performance issues through the following techniques:

- Collecting a wide range of performance data about the operating system on which the application is running.
- Different ways of data processing and visual representation - from general-system view to source view and displaying of processor instructions.
- Detecting probable performance issues and offering ways of their solution.

Intel VTune Performance Analyzer allows the user to collect data about inside events of Intel processors, analyse them and find the most loaded code fragments called hotspots. It also provides features for call graph profiling, gathering time characteristics of calls and detecting fragments that can be parallelized with the highest efficiency by using available processor cores to the full extent.

The tool employs the Sampling technology to evaluate performance issues caused by the most loaded functions. While executing the application, the utility measures time spent in modules, functions, code lines, and samples processor events (those related to branch prediction, efficiency of micro-op fusion, partial outages) associated with a particular module or function.

Thus, you can analyse all the processes in the program and locate hotspots which lead to performance issues. For that purpose, Sampling counts the number of processor ticks, executed instructions, processor cache misses and branch mispredictions. Running a Sampling collection attached to your application, you will get a diagram showing distribution of processor events among modules executed in the system, as well as relative evaluations of the application's performance.

To find out which lines have caused hotspots you just need to drill down to the Source View to see code lines associated with particular events. You can use these data to determine which computations have caused processor events responsible for performance losses and optimize the program in the most efficient way.