1. Write a program to demonstrate merge sort using parallel programing and record the time.

```
#include<stdio.h>
#include<omp.h>
#include<stdlib.h>
void merge(int a[],int l,int mid,int h){
   int n1 = mid-l+1;
   int n2 = h-mid;
   int arr1[n1],arr2[n2];
   for(int i=0;i<n1;i++) arr1[i] = a[l+i];
   for(int i=0;i<n2;i++) arr2[i] = a[mid+1+i];
   int i=0,j=0,k=l;
   while(i<n1 && j<n2){
      if(arr1[i]<=arr2[j])
      {
         a[k++] = arr1[i++];
      }
      else{
         a[k++] = arr2[j++];
      }
   }

   while(i<n1){
      a[k++] = arr1[i++];
   }
   while(j<n2){
      a[k++] = arr2[j++];
   }
}

void mergesortParallel(int a[],int l,int h){
   if(l<h){
      int mid = l+(h-l)/2;
      #pragma omp parallel sections
      {
         #pragma omp section
         mergesortParallel(a,l,mid);

         #pragma omp section
         mergesortParallel(a,mid+1,h);
      }
      merge(a,l,mid,h);
   }
```

```
}

void mergesortSerial(int a[],int l,int h){
   if(l<h){
      int mid = l+(h-l)/2;    {
         mergesortSerial(a,l,mid);
         mergesortSerial(a,mid+1,h);
      }
      merge(a,l,mid,h);
   }
}


void main(){
   int *a,num,i;
         printf("enter number:");
         scanf("%d",&num);
         a = (int *)malloc(sizeof(int)*num);

         // printf("array before sorting");
         // for(i=0;i<num;i++){
         //       a[i]= rand()%100;
         //       printf("%d ",a[i]);
         // }

         double start = omp_get_wtime();
         mergesortSerial(a,0,num-1);
         double end = omp_get_wtime();

   printf("\narray after sorting\n");

         // for(i =0;i<num;i++) printf("%d ",a[i]);

         double val = end - start;
         printf("\nTime for serial is:%f\n",val);

   start = omp_get_wtime();
   mergesortParallel(a,0,num-1);
   end = omp_get_wtime();


   val = end-start;
   printf("Time for parallel execution is %f\n",val);
}
```

2. Write a program to calculate the value of PI parallelly.

```
#include<stdio.h>
#include<omp.h>
#define NUM_THREADS 12          // this line is very important
static long num_steps = 100000; double step;

void main()
{
        int i;
        double x,pi,sum[NUM_THREADS];
        step = 1.0/(double) num_steps;
        #pragma omp parallel private(i,x)
        {
                int id = omp_get_thread_num();
        double local_sum = 0.0;
                for(i = id,sum[id]=0.0;i<num_steps;i+=NUM_THREADS){
                        x = (i+0.5)*step;
                        local_sum+= 4.0/(1.0+x*x);
                }

              #pragma omp critical
              {
                 sum[0] += local_sum;
              }
        }

        pi = sum[0]/num_steps;
        printf("pi = %6.12f\n",pi);
}
```

3. Write a program that divides the iterations into chunks containing 2 iterations respectively. (OMP_SCHEDULE=static,2)

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

//iterns
void main(){
   int iterns,i,itern[8];
   printf("enter iterns:");
   scanf("%d",&iterns);


   #pragma omp parallel for schedule(static,2)
   for(i=1; i<=iterns; i++){
      int t = omp_get_thread_num();
      itern[t]+=1;
      printf("thread %d itern %d value: %d\n",t,itern[t],i);

   }
}
```

4. Calculate fibonacci number using parallel directive.

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
//fib
int fib(int n){
   int i,j;
   if(n<2) return n;
   else{
      #pragma omp task shared(i) firstprivate(n)
      i = fib(n-1);

      #pragma omp task shared(j) firstprivate(n)
       j = fib(n-2);
      #pragma omp taskwait
      return i+j;
   }
}
void main(){
   int n = 20;
   double start = omp_get_wtime();
   #pragma omp parallel for
   for(int i=0;i<n;i++){
      int t = omp_get_thread_num();
      printf("thread: %d fib(%d) = %d\n",t,i,fib(i));
   }
   double end = omp_get_wtime();
   printf("using schedule time is : %f\n",end-start);

}
```

5. Write a program to calculate prime numbers from 1 to n employing parallel directive.

```
//primes in 1-n

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

int isPrime(int n){
  if(n == 0 || n == 1){
    return 0;
  }
  for(int i = 2; i <= n/2; i++){
    if(n%i == 0){
      return 0;
    }
  }
  return 1;
}

int main(){

  int x = 1,n = 50;
  int primes[100];

  double start = omp_get_wtime();
  #pragma omp parallel for
  for(int i = 1; i <= n;i++){
    if(isPrime(i)){
      int t = omp_get_thread_num();
      printf("process : %d , %d \n",t,i);
    }
  }
  double end = omp_get_wtime();
  printf("The time taken is %lf ",end - start);
  // for(int i = 0; i < n; i++){
    //If needed print numbers here from primes[i]
  // }
}
```

6. Write a program to perform parallel vector addition.

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

void vector_addn_parallel(int n,int *res_vector,int *vector_a, int *vector_b){
   #pragma omp parallel for
        for(int i=0; i<n;i++)
        {
                res_vector[i] = vector_a[i]+vector_b[i];
        }
}


void main(){
   int n = 100000;
        int vector_a[100000],vector_b[100000], res_vector[100000];
        for(int i=0;i<n;i++)
        {
                vector_a[i] = rand()%10;
                vector_b[i] = rand()%10;
        }
   double start = omp_get_wtime();
        vector_addn_parallel(n,res_vector,vector_a,vector_b);
   double end = omp_get_wtime();
   printf("Parallel Time %f\n",end-start);
}
```

7. Write a program to calculate the sum of the first 100 numbers using a critical directive.

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


void main(){
   int n =100;
   int sum = 0; // global so should use critical while updating
   #pragma omp parallel for
   for(int i=1;i<=n;i++){
      #pragma omp critical
      sum+=i;
   }
   printf("sum is %d\n",sum);
}
```