1. Why parallel programming ?

Parallel computing are that computers can execute code more efficiently, which can save time and money by sorting through "big data" faster than ever. Parallel programming can also solve more complex problems, bringing more resources to the table

* Parallel computing models the real world
Things don't happen one at a time, waiting for one event to finish before the next one starts.
To crunch numbers on data points in weather, traffic, finance, industry, agriculture, oceans, ice caps, and healthcare, we need parallel computers.
* Saves time - Serial computing forces fast processors to do things inefficiently.
* Saves money - By saving time, parallel computing makes things cheaper. The more efficient use of resources may seem negligible on a small scale.
* Solve more complex or larger problems - Complex problems can be computed faster with parallel computing.
* Leverage remote resources - With parallel computing multiple computers with several cores each can sift through many times more real-time data than serial computers working on their own.

2. What are the different methods to parallelize your code

* Shared memory - For shared memory parallelism, each core is accessing the same memory so there is no need to pass information between different machines.
Threads - They are multiple paths of execution within a single process.

* Distributed memory - Parallel programming for distributed memory parallelism requires passing messages containing information (code, data, etc) between the different nodes.

* Other type of parallel processing -
GPUs - designed for rendering graphics on a computer quickly. This is done by having a large number of simple processing units for massively parallel calculation.

Spark and Hadoop - systems for implementing computations in a distributed memory environment.

4/5

3 Difference between shared memory and distributed parallel computing.

Shared Memory -

* Shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them on avoiding redundant copies.
* Shared memory is an efficient means of passing data b/w program
* Depending on context, programs may run on a single processor on multiple seperate processors.
* Shared memory is a method of inter-process communication i.e., a way of exchanging data b/w programs running at the same time.
* One process will create an area in RAM which other processes can access. It is a method of conserving memory space by directing access to what would ordinarily be copies of a piece of data to a single instance instead, by using virtual memory mappin

Distributed

* The different physical memories are logically shared over a large address space (virtual memory).
* The processes going on in the distributed system access the physical memory through the logically shared address space.
* It has no physical shared memory
* It provides a virtual address space shared between all nodes

Experiment No. :..... Date :.....

Write example programs to demonstrate each of the following directive and/or method :-
Parallel pragma, Parallel for, get-num-threads(), get-thread-number(), num-threads()

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]){
    #pragma omp parallel
    {
        printf("Hello from thread = %d of %d \n",
        omp-get-thread-num(), omp-get-num-threads());
    }
}
```

```
#include <stdio.h>
#include <omp.h>
void main()
{
    int i, sum=0;
    int thread-sum[4];
    omp-set-num-threads(4);
    #pragma omp parallel
    {
        int ID = omp-get-thread-num();
        thread-sum[ID] = 0;
        #pragma omp for
```

Experiment No. ..... Date :.....

```
        for (i = 1; i <= 100; i++)
        {
            thread_sum[ID] += i;
        }
    }

    for (i=0; i<4; i++)
        sum += thread_sum[i];
    printf ("Sum = %d", sum);
}
```

14/10

Experiment No. :..... Date :.....

→ Write an OPENMP program to find prime numbers

```
#include <stdio.h>
#include <omp.h>
main(){
  int prime[1000], i, j, n;
  printf("\nIn order to find prime numbers from 1 to n,
      enter the value of n:");
  scanf("%d", &n);
  for(i=1; i<=n; i++){
    prime[i] = 1; }
  prime[1] = 0;
  for(i=2; i*i<=n; i++){
    #pragma omp parallel for
    for(j= i*i ; j<=n; j=j+1){
      if(prime[j]==1)
        prime[j]=0; }
  }
  printf("\nPrime numbers from 1 to %d are\n", n);
  for(i=2; i<=n; i++){
    if(prime[i]==1){
      printf("%d\t", i);
    }
  }
  printf("\n");
}
```

Experiment No. .....

Date : .....

→ Write an OPENMP program for merge sort.

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

void merge (int array[], int low, int mid, int high) {
    int temp[30], i, j, k, m;
    j = low;
    m = mid + 1;
    for (i = low; j <= mid && m <= high; i++) {
        if (array[j] <= array[m]) {
            temp[i] = array[j];
            j++; }
        else {
            temp[i] = array[m];
            m++; }
    }
    if (j > mid) {
        for (k = m; k <= high; k++) {
            temp[i] = array[k];
            i++; }
    }
    else {
        for (k = j; k <= mid; k++) {
            temp[i] = array[k];
            i++; }
        }
    for (k = low; k <= high; k++)
        array[k] = temp[k];
}
```

Experiment No. :..... Date :.....

```
void mergesort (int array[], int low, int high) {
  int mid;
  if (low < high) {
    mid = (low + high)/2;
    #pragma omp parallel sections num_threads(2)
    {
      #pragma omp section
      {
        mergesort (array, low, mid);
      }
      #pragma omp section
      {
        mergesort (array, mid+1, high);
      }
    }
    merge (array, low, mid, high); }
}

int main() {
  int array[50], i, size;
  printf ("Enter total number of elements : \n");
  scanf ("%d", &size);
  printf ("Enter %d elements : \n", size);
  for(i=0; i< size ; i++) {
    scanf ("%d", &array[i]);
  }
  mergesort (array, 0, size-1);
```

Experiment No. :..... Date :.....

```
    printf ("Sorted elements are as follows: \n");
    for (i=0; i< size; i++)
        printf ("%d", array[i]);
    printf ("\n");
    return 0;
}
```

→ Critical directive for sum of n numbers

```
#include <stdio.h>
#include <omp.h>
int main (int argc, char** argv) {
    int partial_sum, total_sum;
    #pragma omp parallel private(partial_sum) shared (total_sum)
    {
        partial_sum = 0;
        total_sum = 0;
        #pragma omp for
        {
            for(int i= 1; i<= 1000 ; i++) {
                partial_sum += i; }
        }
        #pragma omp critical
        {
            total_sum += partial_sum;
        }
    }
    printf("Total Sum : %d \n", total_sum);
    return 0;
}
```

Date :

Reduction clause for sum of n numbers

```
#include <omp.h>
#include <stdio.h>
int main() {
    int i;
    const int N = 1000;
    int sum = 0;
    #pragma omp parallel for private(i) reduction(+: sum)
    for (i=0; i<N; i++) {
        sum += i;
    }
    printf("Reduction sum = %d (expected %d)\n",
        sum, ((N-1)*N)/2);
}
```

→ Area under the curve using trapezoidal rule

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void Trap(double a, double b, int n, double* global_result
double f(double x);
int main(int argc, char* argv[]) {
    double global_result = 0.0;
    double a, b;
    int n;
    int thread_count;
    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    Scanf("%lf %lf %d", &a, &b, &n);
```

Experiment No. :..... Date :.....

```
#pragma omp parallel num-threads (thread-count)
Trap(a,b,n, &global-result);
printf("Thread number is %d.\n", thread-count);
printf("With n = %d trapezoids, our estimate\n", n);
printf("of the integral from %f to %f = %.14e.\n",
 a,b, global-result);
return 0;
}

void Trap(double a, double b, int n, double* global-result-
   double h, x, my-result;
   double local-a, local-b;
   int i, local-n;
   int my-rank = omp-get-thread-num();
   int thread-count = omp-get-num-threads();
   h = (b-a)/n;
   local-n = n/thread-count;
   local-a = a + my-rank * local-n *h;
   local-b = local-a + local-n * h;
   my-result = (f(local-a) + f(local-b))/2.0;
   for(i=1; i<= local-n-1; i++) {
      x = local-a + i * h;
      my-result += f(x);
   }
   my-result = my-result * h;
   #pragma omp critical
   *global-result-p += my-result;
}
double f(double x) {
   return x*x*x + 2*x + 5;
}
```

14/10

```
if (n % thread_count != 0) {
    fprintf (stderr, "n must be evenly divisible by
        thread_count \n");
    exit (0);
}
```

→ OPENMP program to find value of pi

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void monteCarlo (int N, int K) {
    double x, y;
    double d;
    int pCircle = 0;
    int pSquare = 0;
    int i = 0;
#pragma omp parallel firstprivate (x, y, d, i)
  reduction (+ : pCircle, pSquare) num_threads (K)
{
    srand48 ((int) time (NULL));
    for (i = 0 ; i < N ; i++) {
        x = (double) drand48 ();
        y = (double) drand48 ();
        d = ((x * x) + (y * y));
        if (d <= 1) {
            pCircle++;
        }
        pSquare++;
}
```

Experiment No. :..... Date :.....

```
double pi = 4.0 * ((double) pCircle / (double)(pSquare));
printf("Final estimation of Pi = %f \n", pi);
}

int main() {
    int N = 100000;
    int K = 8;
    monteCarlo(N, K);
}
```

→ Write an OPENMP program to implement sections directive

```
int main() {
    #pragma omp parallel
    {
        # pragma omp section
        printf("This is from thread %d \n",
                omp_get_thread_num());
        # pragma omp section
        printf("This is from thread %d \n",
                omp_get_thread_num());
    }
}
```

Experiment No. : ..... Date : .....

Demonstrate schedule with various parameters/combinations.

Scheduling is a method in OpenMP to distribute iterations to different threads in for loop
There are 4 scheduling techniques -

* Static
* Dynamic
* Guided
* Runtime

```
int main()
{
 X(omp_set_num_threads(4);)X
   #pragma omp parallel for schedule(static,3)
   for(int i=0; i<20; i++)
   {
     printf("Thread %d is running number %d\n",
       omp_get_thread_num(), i);
   }
   return 0;
}
```

```
int main(){
   #pragma omp parallel for schedule(dynamic, 1)
   for(int i=0; i<20; i++){
       printf("Thread %d is running number %d\n",
           omp_get_thread_num(), i); }
   return 0;
}
```

```
int main()
{
    omp_get_num_threads(4);
    #pragma omp parallel for schedule(guided, 3)
    for (int i=0; i<20; i++)
    {
        printf("Thread %d is running number %d \n",
            omp_get_thread_num(), i);
    }

    return 0;
}
```