Reg No. 20BCE1685 ***Name: Priyanshu vyas***

WEEK 3 - PDC- LAB Program Submission

Questions:

---

1. Sum of first n numbers.

Aim

To implement Sum of first n numbers using OpenMp.

Algorithm

Take user input of n.

Start clock for counting execution time.

Declare omp condition.

Inside a for loop from 1 to n compute the sum.

Print sum and execution time for different no of threads.

Program

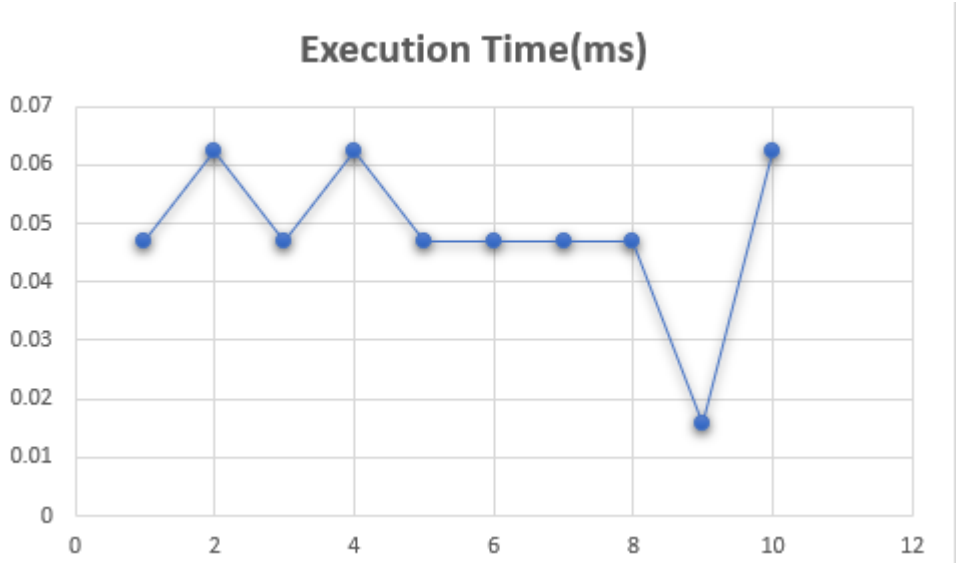
```
#include<stdio.h>
#include<omp.h>
#include<time.h>
int main()
{
   clock_t start,end;
   long int n;
   printf("Enter n: ");
   scanf("%ld",&n);
   int nthreads=1;
```

```
    long int sum=0;
    while(nthreads<=10) {
    start=clock();
    omp_set_num_threads(nthreads);
    #pragma omp parallel for reduction(+:sum)
    for(int i=1;i<=n;i++)
    {
        sum+=i;
    }
    end=clock();
    double exetime=(double)(end-start)/CLOCKS_PER_SEC;
    printf("Execution Time for %d threads is %lf \n",nthreads,exetime);
    nthreads++;
    }
    printf("Sum = %ld \n",sum);
    return 0;
}
```

Output

```
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$ gcc 1.c -fopenmp
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$ ./a.out
Enter n: 10000000
Execution Time for 1 threads is 0.046875
Execution Time for 2 threads is 0.062500
Execution Time for 3 threads is 0.046875
Execution Time for 4 threads is 0.062500
Execution Time for 5 threads is 0.046875
Execution Time for 6 threads is 0.046875
Execution Time for 7 threads is 0.046875
Execution Time for 8 threads is 0.046875
Execution Time for 9 threads is 0.015625
Execution Time for 10 threads is 0.062500
Sum = 500000050000000
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$
```

Graph



Result

Hence the program is successfully implemented and passed execution without errors.

---

2. Sum of n random numbers.

Aim

To implement Sum of first n random numbers using OpenMp.

Algorithm

Take n random numbers.

Start clock for counting execution time.

Declare omp condition.

Inside a for loop from 1 to n compute the sum.

Print sum and execution time for different no of threads.

Program

```
#include<stdio.h>

#include<stdlib.h>

#include<omp.h>
```

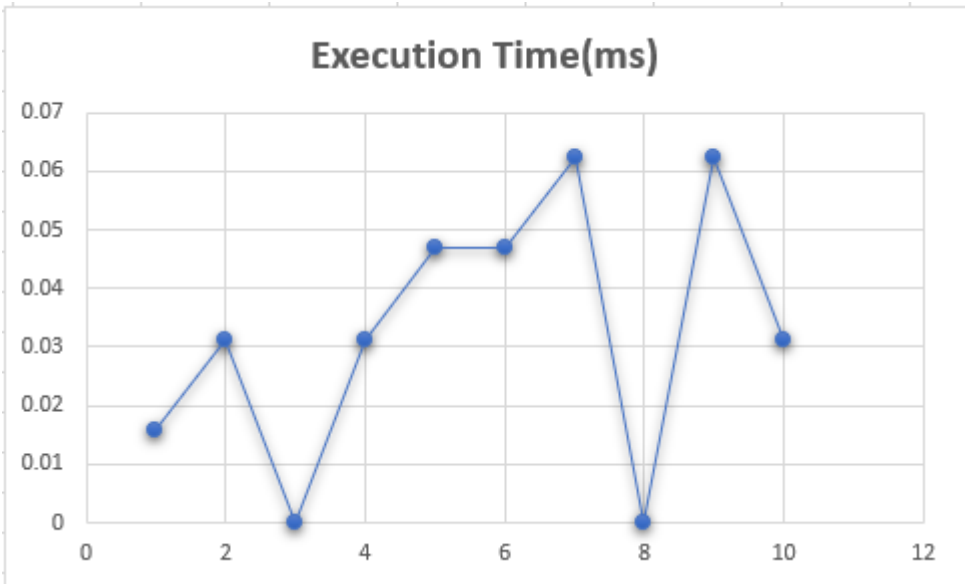
```
#include<time.h>
int main()
{
    clock_t start,end;
    long int n;
    printf("Enter n: ");
    scanf("%ld",&n);
    int nthreads=1;
    long int sum=0;
    while(nthreads<=10) {
    start=clock();
    omp_set_num_threads(nthreads);
    #pragma omp parallel for reduction(+:sum)
    for(int i=1;i<=n;i++)
    {
        sum+=rand();
    }
    end=clock();
    double exetime=(double)(end-start)/CLOCKS_PER_SEC;
    printf("Execution Time for %d threads is %lf \n",nthreads,exetime);
    printf("Sum = %ld \n",sum);
    nthreads++;
    }
    return 0;
```

}

Output

```
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$ gcc 2.c -fopenmp
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$ ./a.out
Enter n: 100000
Execution Time for 1 threads is 0.015625
Sum = 107345864261546
Execution Time for 2 threads is 0.031250
Sum = 214690051225657
Execution Time for 3 threads is 0.000000
Sum = 322177984143889
Execution Time for 4 threads is 0.031250
Sum = 429770314740706
Execution Time for 5 threads is 0.046875
Sum = 537095867682884
Execution Time for 6 threads is 0.046875
Sum = 644303575336424
Execution Time for 7 threads is 0.062500
Sum = 751717033804613
Execution Time for 8 threads is 0.000000
Sum = 859064561043479
Execution Time for 9 threads is 0.062500
Sum = 966140731467791
Execution Time for 10 threads is 0.031250
Sum = 1073756018481283
Sum = 1073756018481283
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$
```

Graph



Result

Hence the program is successfully implemented and passed execution without errors.

---

3. Addition of 2 integer arrays.

Aim

To implement Two Array’s Addition using OpenMp.

Algorithm

Take user input of Two Arrays a, b of size n.

Start clock for counting execution time.

Declare openmp condition.

Inside a for loop i from 1 to n compute the sum of a[i] and b[i] in new array c.

Print Resultant Array and execution time for different no of threads.

Program

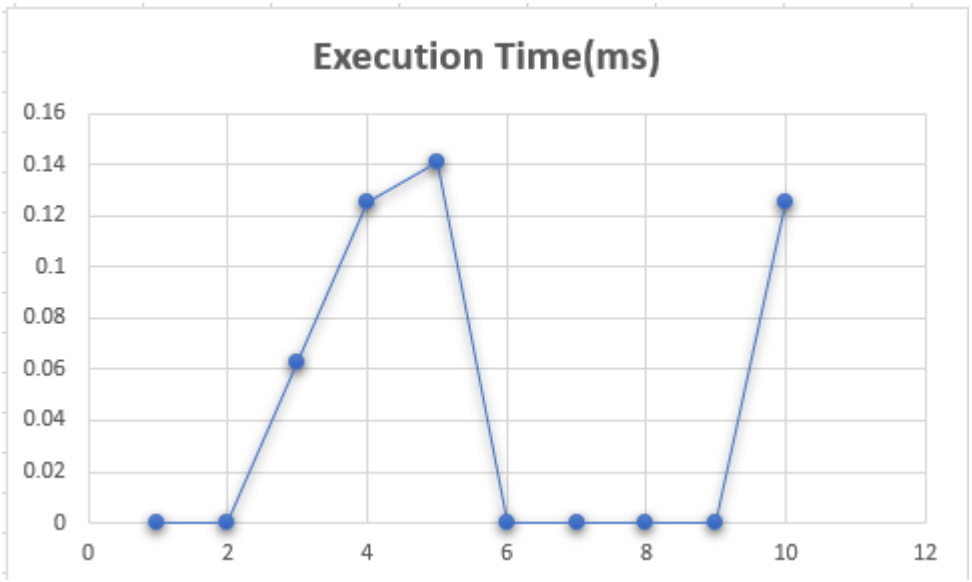
```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
#include<time.h>
#include<unistd.h>
int main()
{
   clock_t start,end;
   long int n;
   printf("Enter n: ");
   scanf("%ld",&n);
   int nthreads=1;
   int a[n],b[n],c[n];
   for(int i=0;i<n;i++)
      scanf("%d",&a[i]);
   for(int i=0;i<n;i++)
```

```
    scanf("%d",&b[i]);
  int sum;
  while(nthreads<=10) {
  start=clock();
  omp_set_num_threads(nthreads);
  sleep(4);
  #pragma omp parallel for reduction(+:sum)
  for(int i=0;i<n;i++)
  {
      sum=a[i]+b[i];
    c[i]=sum;
  }
  end=clock();
  double exetime=((double)(end-start))/CLOCKS_PER_SEC;
  printf("Execution Time for %d threads is %f \n",nthreads,exetime);
  nthreads++;
  }
  for(int i=0;i<n;i++)
    printf("%d",c[i]);
  printf("\n");
  return 0;
}
```

Output

```
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$ gcc -fopenmp 3.c
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$ ./a.out
Enter n: 5
1 2 3 4 5
1 2 3 4 5
Execution Time for 1 threads is 0.000000
Execution Time for 2 threads is 0.000000
Execution Time for 3 threads is 0.062500
Execution Time for 4 threads is 0.125000
Execution Time for 5 threads is 0.140625
Execution Time for 6 threads is 0.000000
Execution Time for 7 threads is 0.000000
Execution Time for 8 threads is 0.000000
Execution Time for 9 threads is 0.000000
Execution Time for 10 threads is 0.031250
246810
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$
```

Graph



Result

Hence the program is successfully implemented and passed execution without errors.

---

4. Identify the maximum element in a list of thousand numbers.

Aim

To implement maximum element in a list of thousand numbers using OpenMp.

Algorithm

Use rand() function to take 1000 numbers.

Start clock for counting execution time.

Declare openmp condition.

Inside a for loop i from 0 to n find the maximum element from list of 1000 elements.

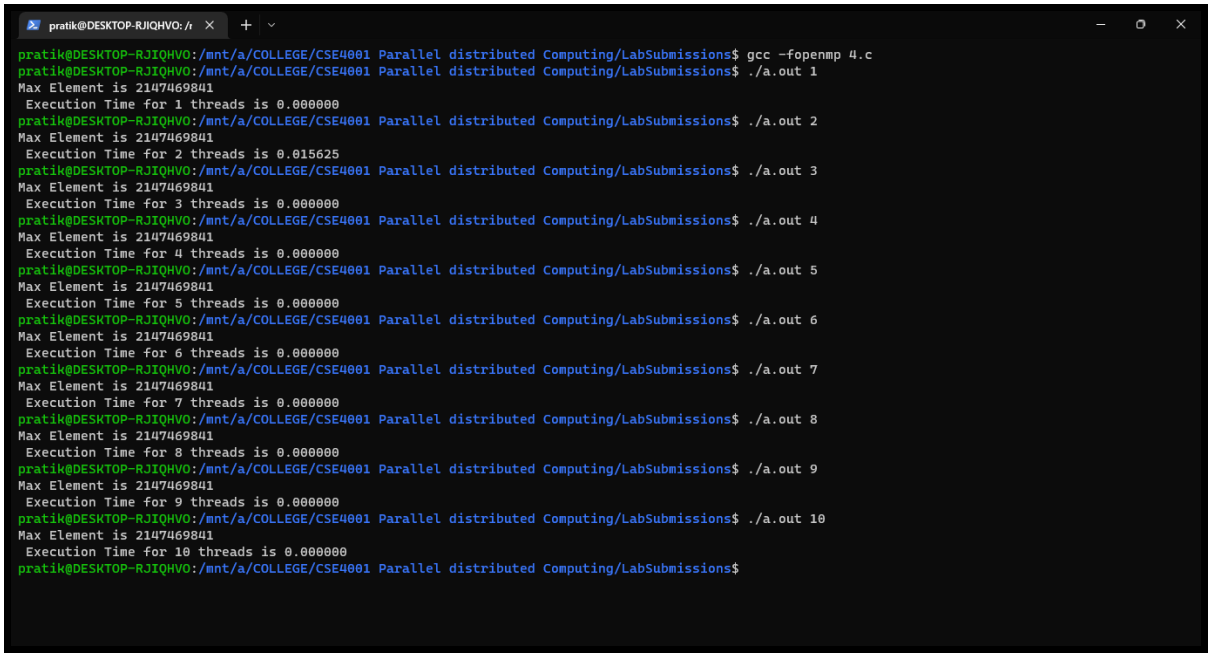
Print Maximum Element and execution time for different no. of threads.

Program

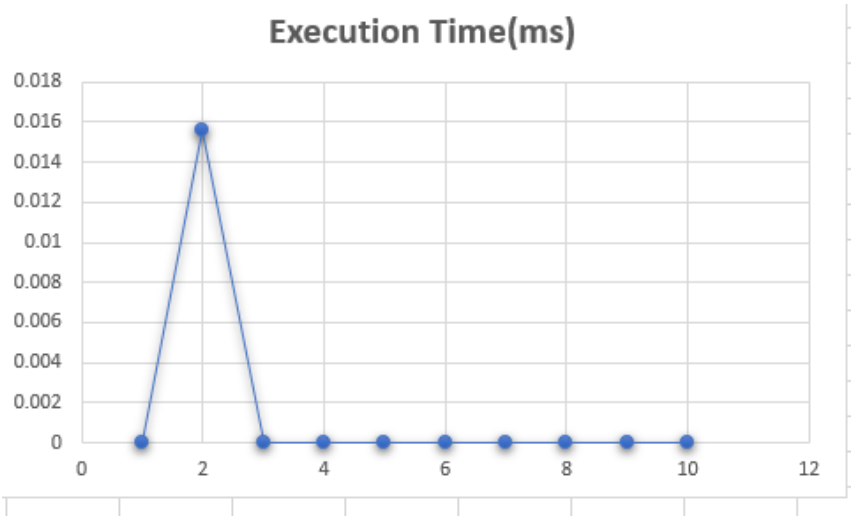
```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
#include<time.h>
#include<limits.h>
int main(int argc, char *argv[])
{
   clock_t start,end;
   int max=INT_MIN;
   start=clock();
   int nthreads=atoi(argv[1]);
   omp_set_num_threads(nthreads);
   #pragma omp parallel for reduction(max:max)
   for(int i=0;i<1000;i++)
   {
      int k=rand();
         if(max<k)
         max=k;
   }
   end=clock();
```

```
    double exetime=((double)(end-start))/CLOCKS_PER_SEC;
    printf("Max Element is %d \n Execution Time for %d threads is %f \n",max,nthreads,exetime);
    return 0;
}
```

Output



Graph



Result

Hence the program is successfully implemented and passed execution without errors.

---

5. Identify the minimum value in a list of 10,000 numbers.

Aim

To implement the minimum value in a list of 10,000 numbers using OpenMp.

Algorithm

Use rand() function to take 1000 numbers.

Start clock for counting execution time.

Declare openmp condition.

Inside a for loop i from 0 to n find the maximum element from list of 1000 elements.

Print Minimum Element and execution time for different no. of threads.

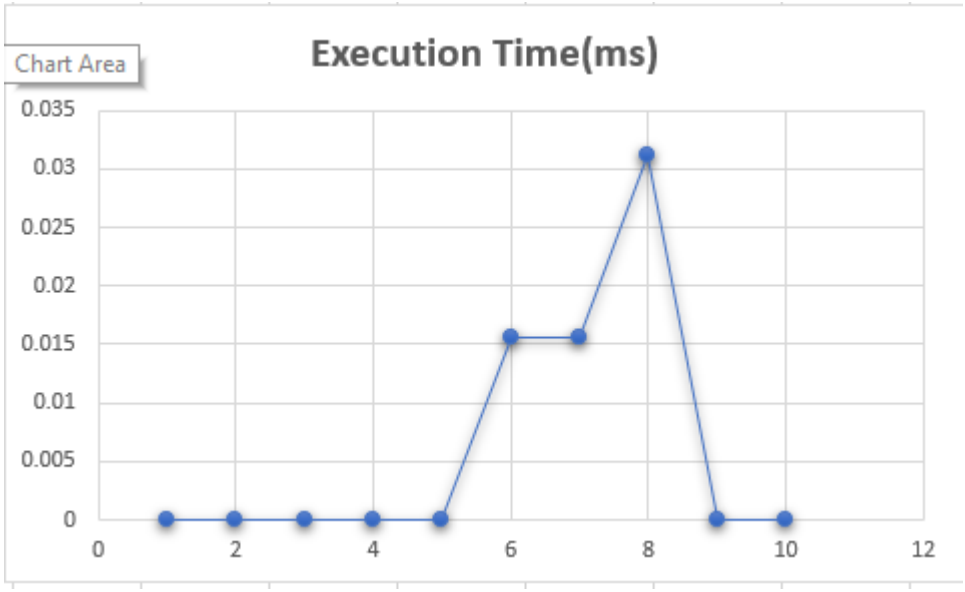
Program

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
#include<time.h>
#include<limits.h>
int main(int argc, char *argv[])
{
   clock_t start,end;
   int min=INT_MAX;
   start=clock();
   int nthreads=atoi(argv[1]);
   omp_set_num_threads(nthreads);
```

```
    #pragma omp parallel for reduction(min:min)
    for(int i=0;i<10000;i++)
    {
       int k=rand();
          if(min>k)
          min=k;
    }
    end=clock();
    double exetime=((double)(end-start))/CLOCKS_PER_SEC;
    printf("Min Element is %d \n Execution Time for %d threads is %f \n",min,nthreads,exetime);
    return 0;
}
```

Output

Graph



Result

Hence the program is successfully implemented and passed execution without errors.

---

6. Perform Matrix Vector Multiplication.

Aim

To implement Matrix Vector Multiplication using OpenMp.

Algorithm

Take input of Matrix and vector.

Start clock for counting execution time.

Declare openmp condition.

Inside a nested for loop i, j from 1 to n compute the matrix multiplication of matrix and vector.

Print Resultant vector and execution time for different no of threads.

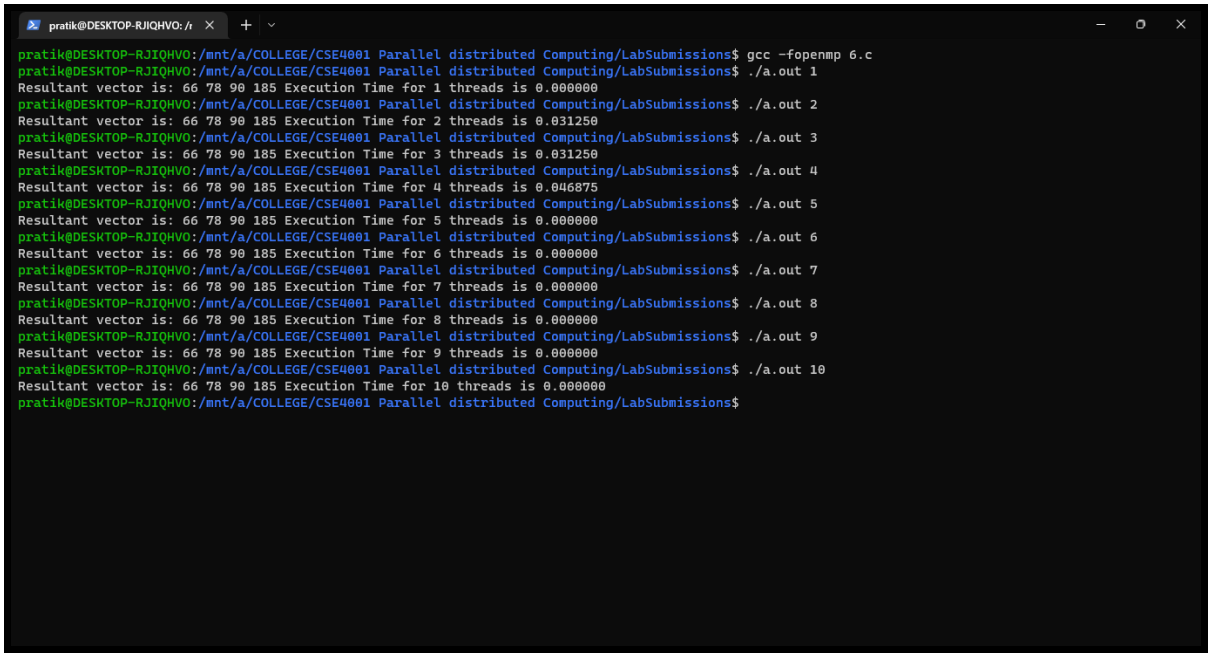
Program

```
#include<stdio.h>

#include<stdlib.h>

#include<omp.h>
```

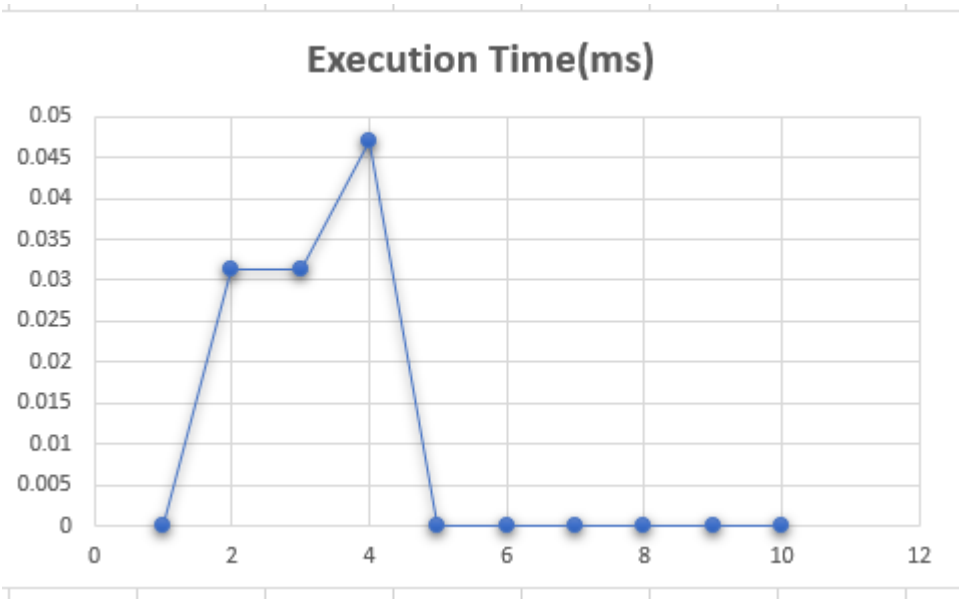
```
#include<time.h>
#include<limits.h>
#include<unistd.h>
int main(int argc, char *argv[])
{
   clock_t start,end;
   int matrix[4][4]={{1,4,7},{2,5,8},{3,6,9},{10,14,17}};
   int vector[4]={1,4,7};
   int result[4]={1,4,7};
   start=clock();
   int nthreads=atoi(argv[1]);
   omp_set_num_threads(nthreads);
   int sum=0;
   #pragma omp parallel for reduction(+:sum)
   for(int i=0;i<4;i++)
   {
      sum=0;
      for(int j=0;j<4;j++)
      {
         sum+=matrix[i][j]*vector[j];
      }
      result[i]=sum;
   }
   sleep(2);
```

```
    printf("Resultant vector is: ");
    for(int i=0;i<4;i++)
    {
        printf("%d ",result[i]);
    }
    end=clock();
    double exetime=((double)(end-start))/CLOCKS_PER_SEC;
    printf("Execution Time for %d threads is %f \n",nthreads,exetime);
    return 0;
}
```

Output



Graph



Result

Hence the program is successfully implemented and passed execution without errors.

---

7. Write multiple programs to illustrate ||ism in nested for, barrier and nowait.

Aim

To implement nested loop, barrier and no wait parallelism using OpenMp.

Algorithm

Nested loop:

Take two array input ‘a’, ‘b’.

Start the clock();

Declare the open mp condition of/for nested loop.

Add the two arrays and store in a new array ‘c’.

End the clock and print execution time.

Print the result stored in c.

Barrier:

Set no of threads.

Start the clock.

Define #pragma omp condition for parallel computation.

Print a statement for showing before barrier condition.

Use #pragma omp barrier.

Print a statement in barrier to show Impact of barrier.

End the clock and Print execution time.

Nowait:

Define an array's 'a', 'b', 'z', 'y'.

Take inputs of a and z from user.

Start the clock.

Define open mp condition accordingly.

Compute the sum of two consecutive elements of a in b.

Compute square roots of elements of 'z' in y.

End the clock and print the execution time.

Print the resultant arrays b and y.

Program

Nested loop:

```
// Nested for loop parallelism

#include <stdio.h>

#include <omp.h>

#include <time.h>

#include<stdlib.h>

#include<unistd.h>
```

```
int main(int argc,char *argv[])
{
    int a[3][3], b[3][3], c[3][3];
    printf("Enter a 3*3 Matrix : \n");
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            scanf("%d", &a[i][j]);
    printf("Enter another 3*3 Matrix : \n");
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            scanf("%d", &b[i][j]);

    clock_t start,end;
    int nthreads=atoi(argv[1]);
    while(nthreads>0)
    {
        start=clock();
        sleep(2);
        #pragma omp parallel for
        for (int i = 0; i < 3; i++)
        {
            #pragma omp parallel for
            for (int j = 0; j < 3; j++)
                c[i][j] = a[i][j] + b[i][j];
```

```
        }
        end=clock();
        double exetime=(double)(end-start)/CLOCKS_PER_SEC;
        printf("Execution time of %d threads is %lf \n",nthreads,exetime);
        nthreads--;
    }
    printf("Addition of 2 - 3*3 Matrices are : \n");
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
            printf("%d ", c[i][j]);
        printf("\n");
    }
    return 0;
}
```

Barrier:

```
// barrier parallelism
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include<time.h>

int main(int argc,char *argv[])
{
```

```
	int nthreads=atoi(argv[1]);
	while(nthreads>0)
	{
		omp_set_num_threads(nthreads);
		clock_t start,end;
		start=clock();
		#pragma omp parallel
		{
			printf("[Thread %d] I print my first message.\n", omp_get_thread_num());
			#pragma omp barrier
			#pragma omp single
			{
				printf("The barrier is complete, which means all threads have printed their first message.\n");
			}
			printf("[Thread %d] I print my second message.\n", omp_get_thread_num());
		}
		end=clock();
		double exetime=(double)(end-start)/CLOCKS_PER_SEC;
		printf("Executiton time of %d threads is %lf \n \n",nthreads,exetime);
		nthreads--;
	}
}
```

Nowait:

// Nowait parallelism

// If there are multiple independent loops within a parallel region, you can use the nowait clause to avoid the implied barrier at the end of the loop construct.

```
#include <stdio.h>

#include<stdlib.h>

#include <omp.h>

#include <time.h>

#include <math.h>

#include<unistd.h>


void nowait(int n, int m, float *a, float *b, float *y, float *z)

{

   int i;

#pragma omp parallel

   {

#pragma omp for nowait

      for (i = 1; i < n; i++)

         b[i] = (a[i] + a[i - 1]) / 2.0;


#pragma omp for nowait

      for (i = 0; i < m; i++)

         y[i] = sqrt(z[i]);

   }

}
```

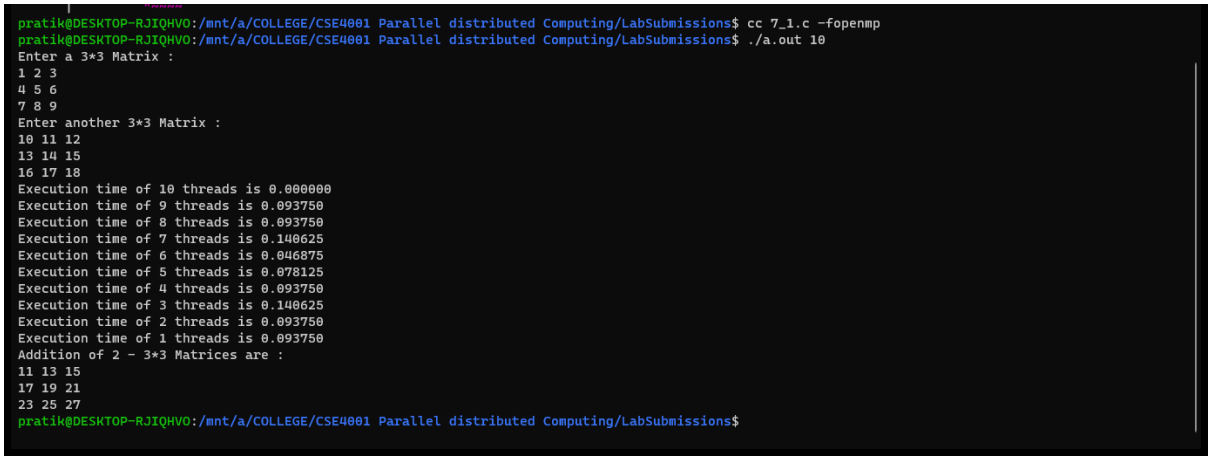
```
int main(int argc,char *argv[])
{
    int n;
    printf("Enter size of Array 'a' :");
    scanf("%d", &n);
    float a[n], b[n];
    printf("Enter elements of Array 'a' :");
    for (int i = 0; i < n; i++)
    {
        scanf("%f", &a[i]);
        b[i] = a[i];
    }
    int m;
    printf("Enter size of Array 'z' :");
    scanf("%d", &m);
    float z[m], y[m];
    printf("Enter elements of Array 'z' :");
    for (int i = 0; i < m; i++)
    {
        scanf("%f", &z[i]);
        y[i] = z[i];
    }
    int nthreads=atoi(argv[1]);
```

```
while(nthreads>0)
{
   omp_set_num_threads(nthreads);
   clock_t start,end;
   start=clock();
   sleep(7);
   nowait(n, m, a, b, y, z);
   end=clock();
   double exetime = (double)(end-start)/CLOCKS_PER_SEC;
   printf("\nExecution time of %d threads is: %f \n",nthreads,exetime);
   nthreads--;
}
printf("Sum of consecutive two elements in a array 'a' is: ");
for (int i = 0; i < n; i++)
{
   printf("%f ", b[i]);
}
printf("\nSquare root of array 'z' are: ");
for (int i = 0; i < m; i++)
{
   printf("%f ", y[i]);
}
printf("\n");
return 0;
```

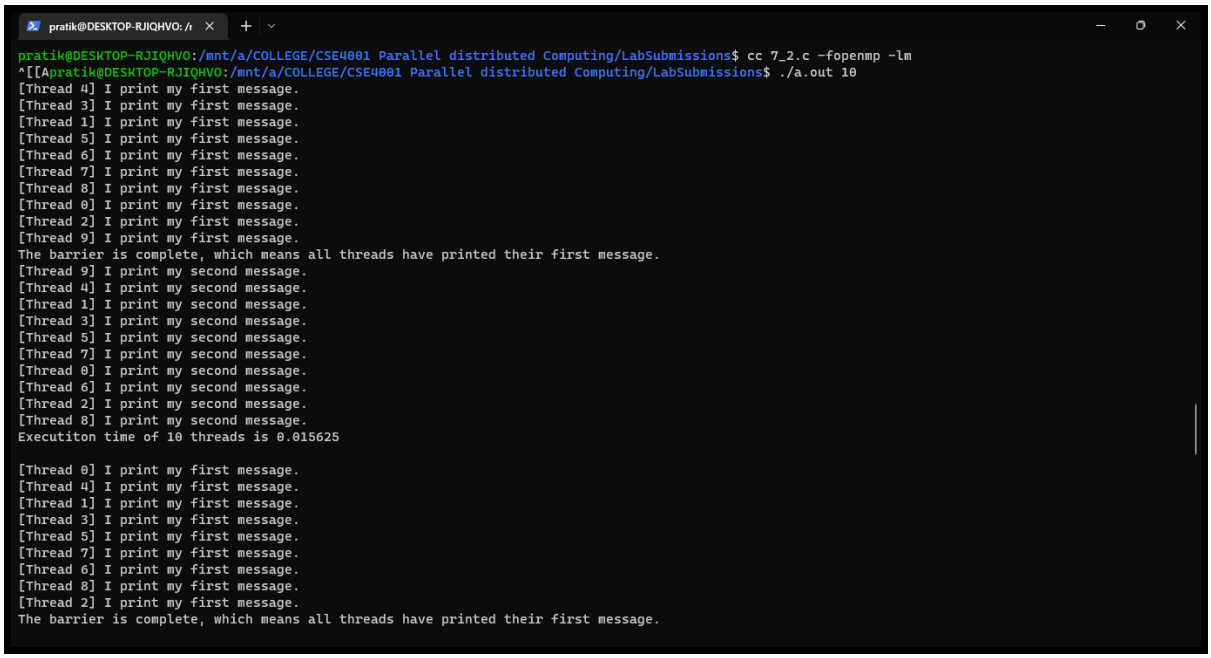
}

Output

Nested loop:



Barrier:



pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$ cc 7_2.c -fopenmp -lm

^[[Apratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$ ./a.out 10

[Thread 4] I print my first message.

[Thread 3] I print my first message.

[Thread 1] I print my first message.

[Thread 5] I print my first message.

[Thread 6] I print my first message.

[Thread 7] I print my first message.

[Thread 8] I print my first message.

[Thread 0] I print my first message.

[Thread 2] I print my first message.

[Thread 9] I print my first message.

The barrier is complete, which means all threads have printed their first message.

[Thread 9] I print my second message.

[Thread 4] I print my second message.

[Thread 1] I print my second message.

[Thread 3] I print my second message.

[Thread 5] I print my second message.

[Thread 7] I print my second message.

[Thread 0] I print my second message.

[Thread 6] I print my second message.

[Thread 2] I print my second message.

[Thread 8] I print my second message.

Executiton time of 10 threads is 0.015625

[Thread 0] I print my first message.

[Thread 4] I print my first message.

[Thread 1] I print my first message.

[Thread 3] I print my first message.

[Thread 5] I print my first message.

[Thread 7] I print my first message.

[Thread 6] I print my first message.

[Thread 8] I print my first message.

[Thread 2] I print my first message.

The barrier is complete, which means all threads have printed their first message.

[Thread 2] I print my second message.

[Thread 0] I print my second message.

[Thread 4] I print my second message.

[Thread 1] I print my second message.

[Thread 3] I print my second message.

[Thread 5] I print my second message.

[Thread 8] I print my second message.

[Thread 7] I print my second message.

[Thread 6] I print my second message.

Executiton time of 9 threads is 0.000000

[Thread 0] I print my first message.

[Thread 6] I print my first message.

[Thread 2] I print my first message.

[Thread 4] I print my first message.

[Thread 1] I print my first message.

[Thread 5] I print my first message.

[Thread 3] I print my first message.

[Thread 7] I print my first message.

The barrier is complete, which means all threads have printed their first message.

[Thread 7] I print my second message.

[Thread 0] I print my second message.

[Thread 2] I print my second message.

[Thread 6] I print my second message.

[Thread 4] I print my second message.

[Thread 1] I print my second message.

[Thread 3] I print my second message.

[Thread 5] I print my second message.

Executiton time of 8 threads is 0.000000

[Thread 0] I print my first message.

[Thread 5] I print my first message.

[Thread 2] I print my first message.

[Thread 6] I print my first message.

[Thread 4] I print my first message.

[Thread 1] I print my first message.

[Thread 3] I print my first message.

The barrier is complete, which means all threads have printed their first message.

[Thread 3] I print my second message.

[Thread 0] I print my second message.

[Thread 2] I print my second message.

[Thread 5] I print my second message.

[Thread 6] I print my second message.

[Thread 4] I print my second message.

[Thread 1] I print my second message.

Executiton time of 7 threads is 0.000000

[Thread 0] I print my first message.

[Thread 1] I print my first message.

[Thread 3] I print my first message.

[Thread 2] I print my first message.

[Thread 5] I print my first message.

[Thread 4] I print my first message.

The barrier is complete, which means all threads have printed their first message.

[Thread 4] I print my second message.

[Thread 0] I print my second message.

[Thread 3] I print my second message.

[Thread 1] I print my second message.

[Thread 2] I print my second message.

[Thread 5] I print my second message.

Executiton time of 6 threads is 0.000000

[Thread 0] I print my first message.

[Thread 4] I print my first message.

[Thread 3] I print my first message.

[Thread 1] I print my first message.

[Thread 2] I print my first message.

The barrier is complete, which means all threads have printed their first message.

[Thread 2] I print my second message.

[Thread 0] I print my second message.

[Thread 3] I print my second message.

[Thread 4] I print my second message.

[Thread 1] I print my second message.

Executiton time of 5 threads is 0.000000

[Thread 0] I print my first message.

[Thread 2] I print my first message.

[Thread 3] I print my first message.

[Thread 1] I print my first message.

The barrier is complete, which means all threads have printed their first message.

[Thread 0] I print my second message.

[Thread 1] I print my second message.

[Thread 3] I print my second message.

[Thread 2] I print my second message.

Executiton time of 4 threads is 0.000000

[Thread 1] I print my first message.

[Thread 2] I print my first message.

[Thread 0] I print my first message.

The barrier is complete, which means all threads have printed their first message.

[Thread 1] I print my second message.

[Thread 0] I print my second message.

[Thread 2] I print my second message.

Executiton time of 3 threads is 0.000000

[Thread 1] I print my first message.

[Thread 0] I print my first message.

The barrier is complete, which means all threads have printed their first message.

[Thread 0] I print my second message.

[Thread 1] I print my second message.

Executiton time of 2 threads is 0.000000

[Thread 0] I print my first message.

The barrier is complete, which means all threads have printed their first message.

[Thread 0] I print my second message.

Executiton time of 1 threads is 0.000000

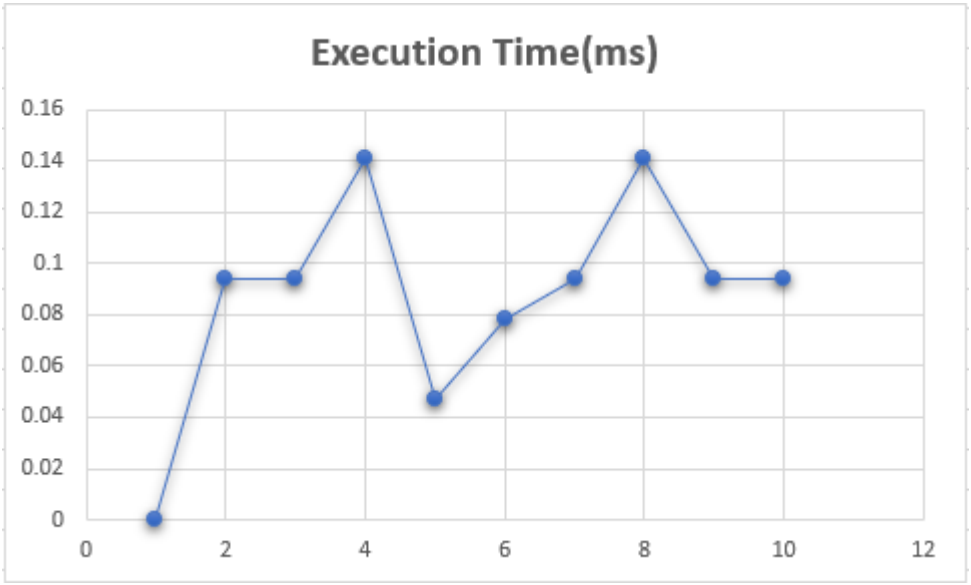
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$

Nowait:

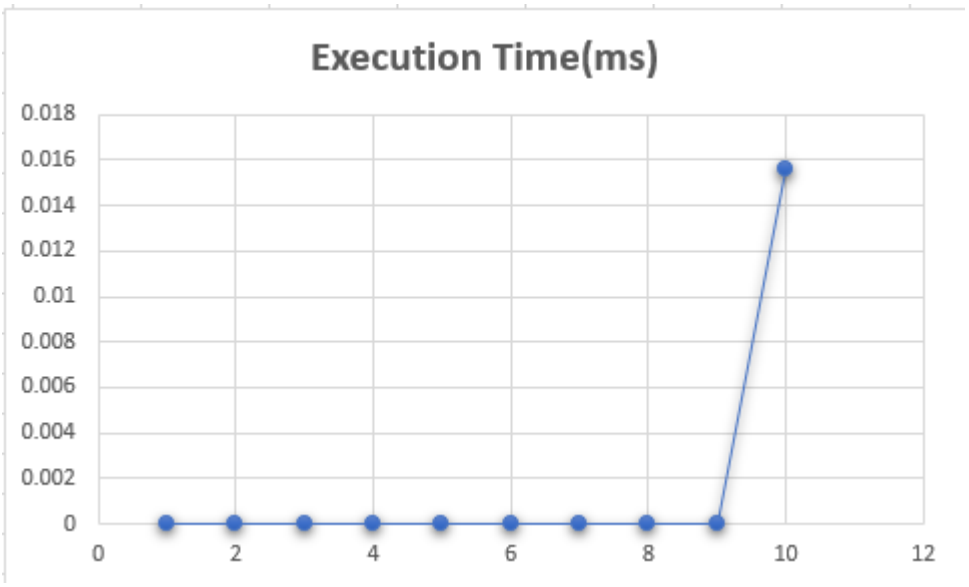
```
pratik@DESKTOP-RJIQHVO: /r
Enter elements of Array 'a' :1 2 3 4
Enter size of Array 'z' :7
Enter elements of Array 'z' :1 2 3 4 5 6 7
Sum of consecutive two elements in a array 'a' is: 1.000000 1.500000 2.500000 3.500000
Square root of array 'z' are: 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
Execution time of 2 threads is: 0.000000
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$ cc 7_3.c -fopenmp -lm
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$ ./a.out 10
Enter size of Array 'a' :4
Enter elements of Array 'a' :1 2 3 4
Enter size of Array 'z' :7
Enter elements of Array 'z' :1 2 3 4 5 6 7

Execution time of 10 threads is: 0.000000

Execution time of 9 threads is: 0.000000

Execution time of 8 threads is: 0.000000

Execution time of 7 threads is: 0.000000

Execution time of 6 threads is: 0.000000

Execution time of 5 threads is: 0.000000

Execution time of 4 threads is: 0.000000

Execution time of 3 threads is: 0.125000

Execution time of 2 threads is: 0.062500

Execution time of 1 threads is: 0.031250
Sum of consecutive two elements in a array 'a' is: 1.000000 1.500000 2.500000 3.500000
Square root of array 'z' are: 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
pratik@DESKTOP-RJIQHVO:/mnt/a/COLLEGE/CSE4001 Parallel distributed Computing/LabSubmissions$
```

Graph

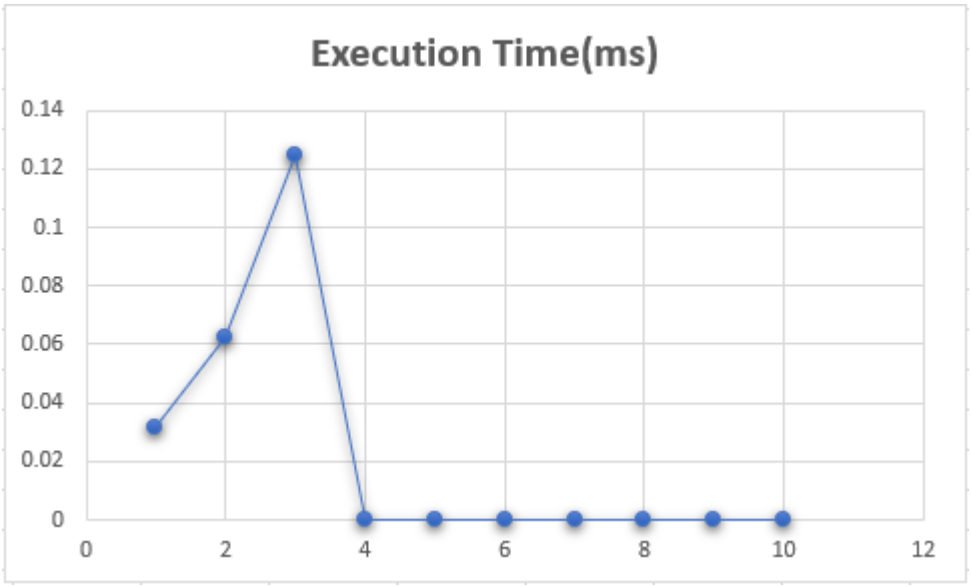
Nested loop:



Barrier:



Nowait:



Result

Hence the programs are successfully implemented and passed execution without errors.

Reg No. 20BCE1685
Name: Priyanshu Vyas
Course code: CSE4001
Course: Parallel and distributing computing
Title: Week 4 Lab Assignment

Q1: Write an open mp program to sort 1000 numbers.

Algorithm:

(I) store 1000 random elements in an array.
(II) use 2 nested loops to sort the elements.
(III) use #pragma omp parallel for collapse(2).
(IV) Compute and print execution time.
(V) print the sorted array.

```
#include<omp.h>

int main(int argc,char *argv[])
{
  clock_t t;
  int list[1000];
  for(int i=0;i<1000;i++)
  {
    list[i]=rand()%10000;
  }
  t=clock();
  #pragma omp parallel for collapse(2)
  for(int i=0;i<1000;i++)
  {
    for(int j=i+1;j<1000;j++)
    {
      if(list[i]>list[j])
      {
        int temp=list[i];
        list[i]=list[j];
        list[j]=temp;
      }
    }
  }
  t=clock()-t;
  for(int i=0;i<1000;i++)
  {
    printf("%d ",list[i]);
  }
  printf("\n\n Execution Time: %lfms \n\n",(double)t/CLOCKS_PER_SEC);
}
```

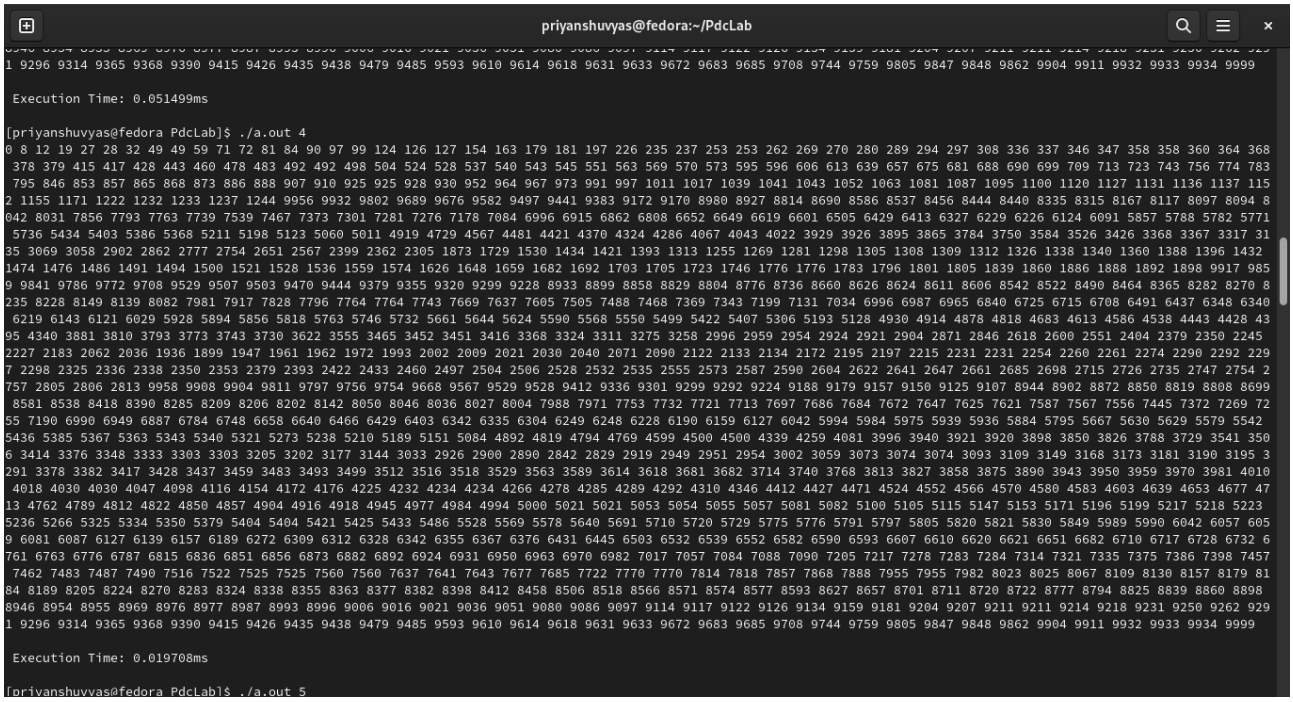
Output:

For Thread 1:

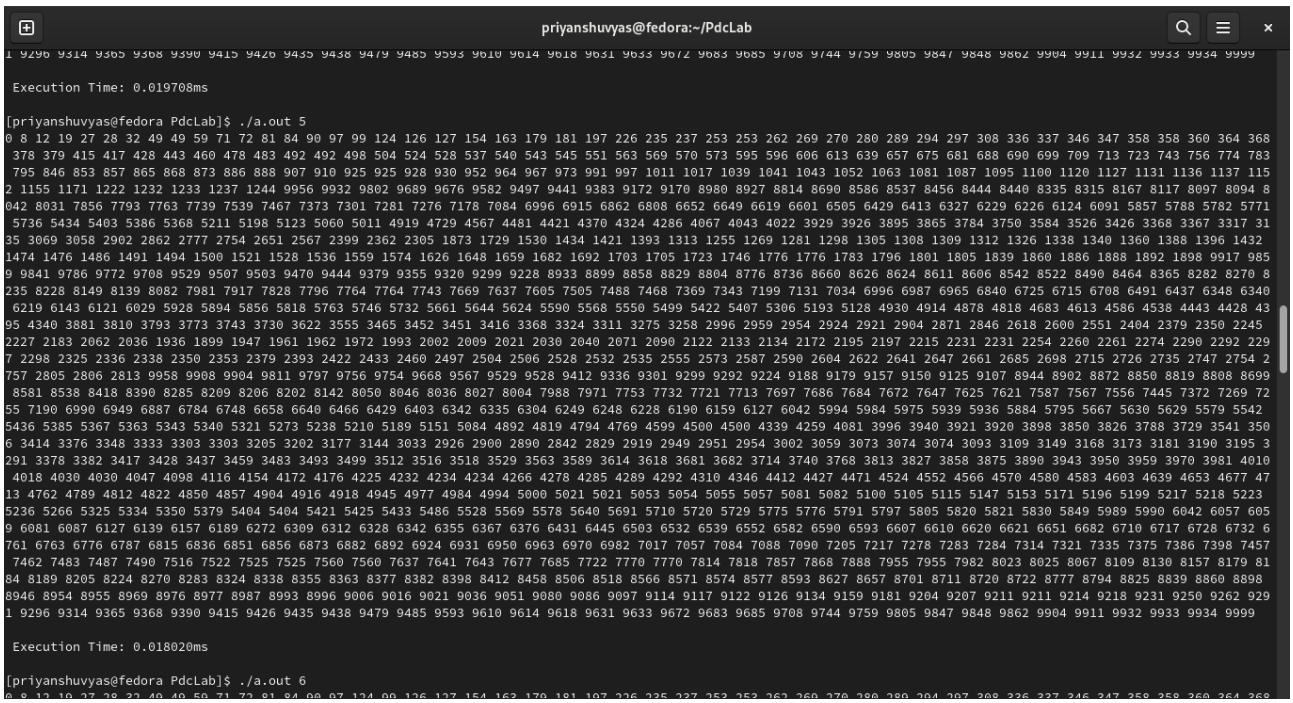
For Thread 2:

For Thread 3:

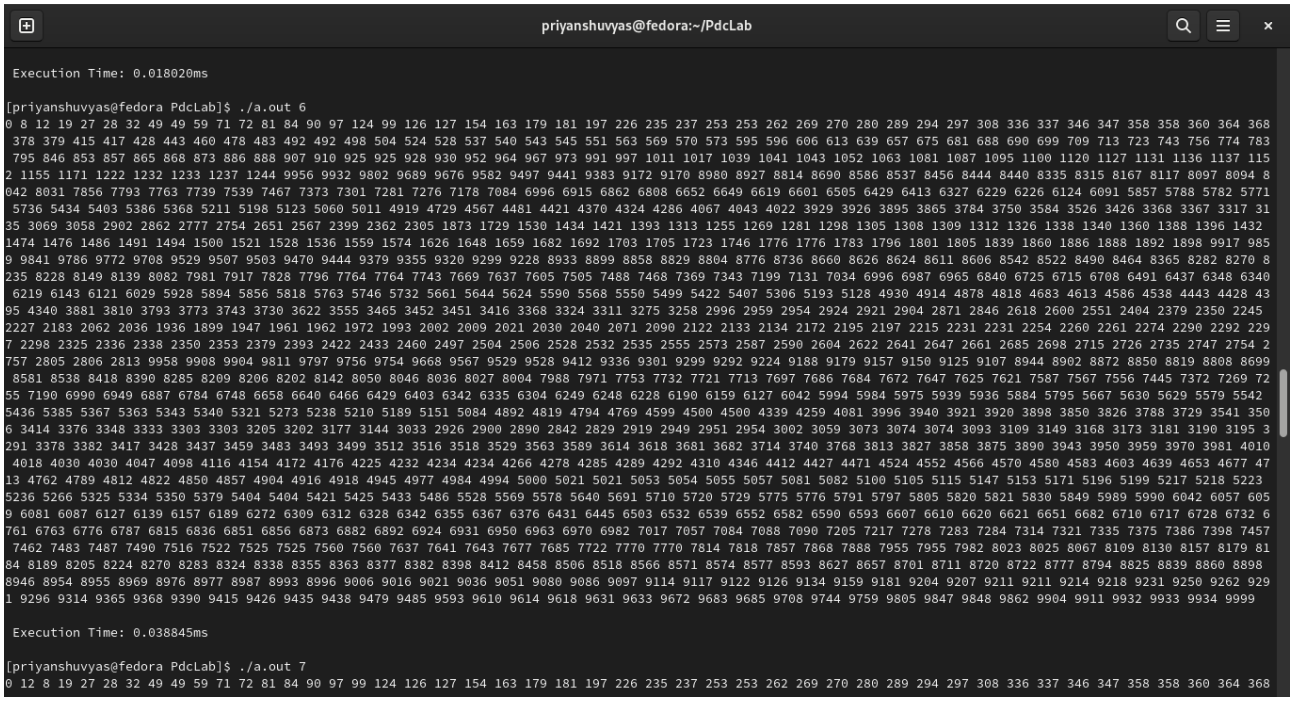
For Thread 4:



For Thread 5:



For Thread 6:



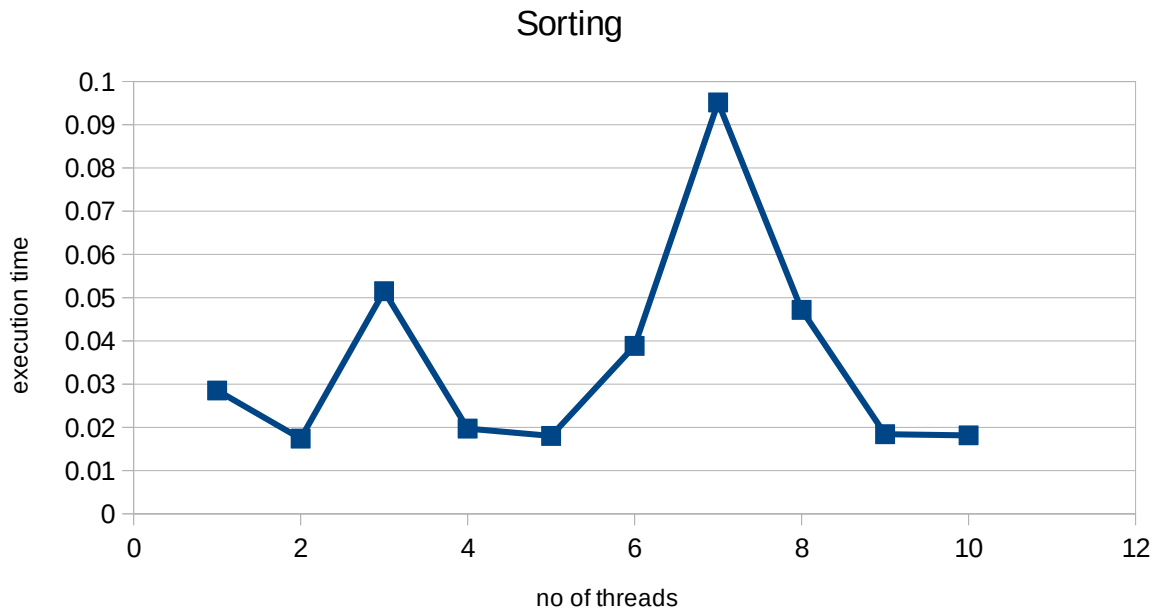
For Thread 7:

For Thread 8:

For Thread 9:

For Thread 10:

Graph:



Result:

Hence all programs are implemented and executed successfully.

Q2: Write an open mp program to find an element in 1000 elements.

Algorithm:

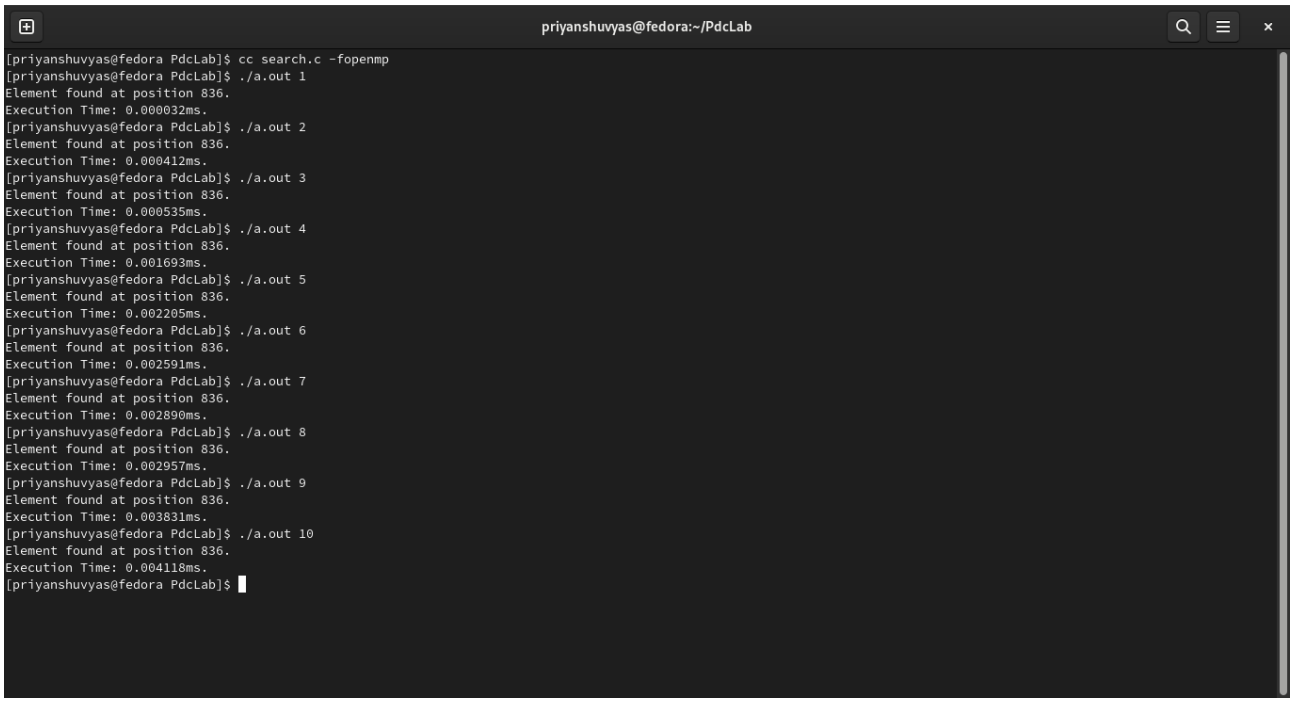
(I) store 1000 random elements in an array.
(II) use a loop to search the element in an array.
(III) use #pragma omp parallel for.
(IV) Compute and print execution time.
(V) print the element index or not found.

Code:

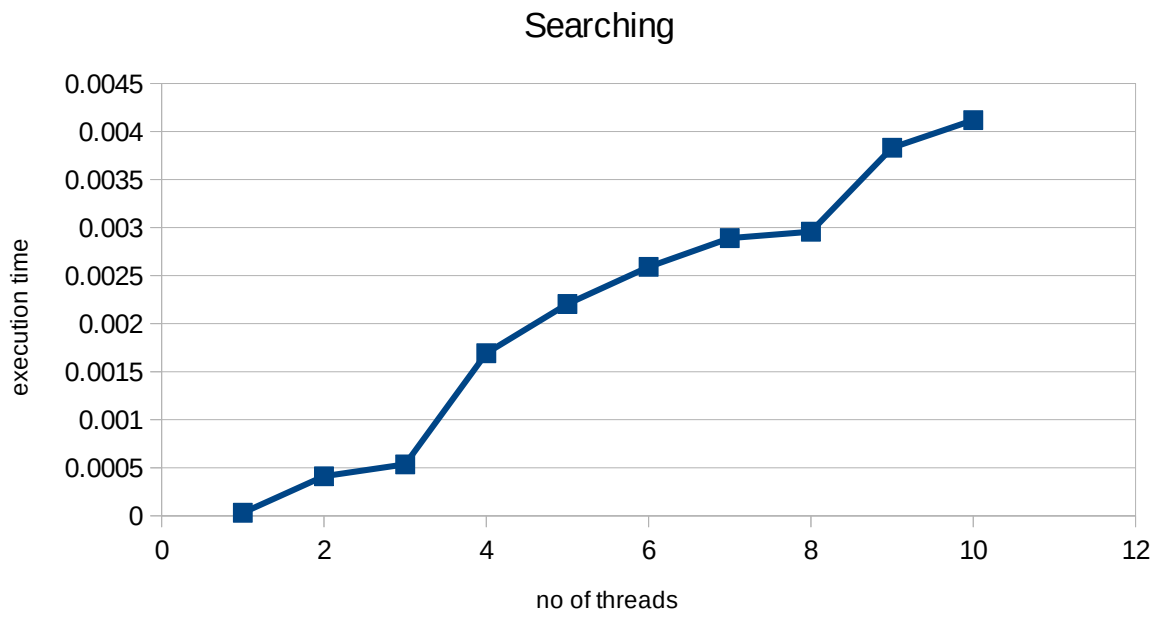
```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>

int main(int argc,char *argv[])
{
  omp_set_num_threads(atoi(argv[1]));
  clock_t t;
  int list[1000];
  for(int i=0;i<1000;i++)
  {
    list[i]=rand()%2000;
  }
  t=clock();
  int flag=0;
```

```
 #pragma omp parallel for
 for (int i=0;i<1000;i++)
 {
  if(list[i]==10)
  {
    printf("Element found at position %d. \n",i+1);
    flag=1;
  }
 }
 if(flag==0)
 {
  printf("Element not found in the list. \n");
 }
 t=clock()-t;
 printf("Execution Time: %lfms. \n",(double)t/CLOCKS_PER_SEC);
}
```

Output:



Graph:



Result:

Hence all programs are implemented and executed successfully.

Q3: Write an open mp program to implement:

(i) Master:

Algorithm:

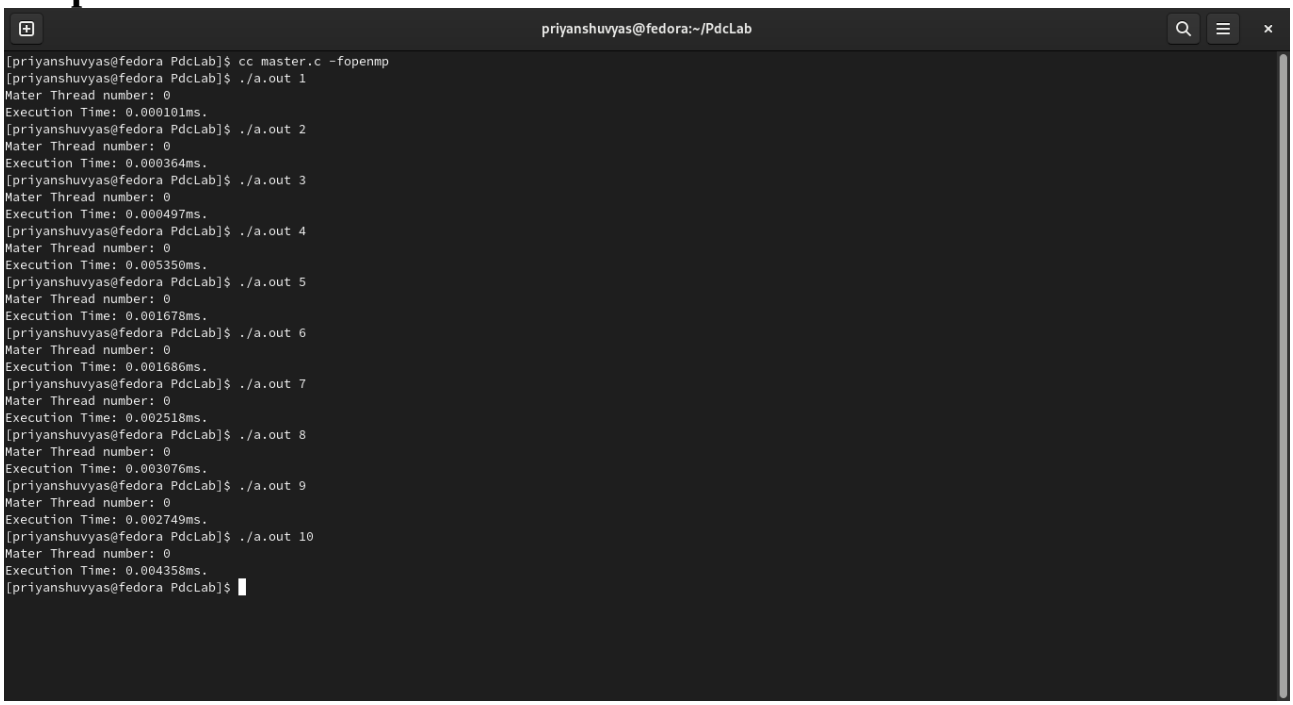
(I) use #pragma omp parallel
(II) use #pragma omp master to print master thread.
(III) Compute and print execution time.

Code:

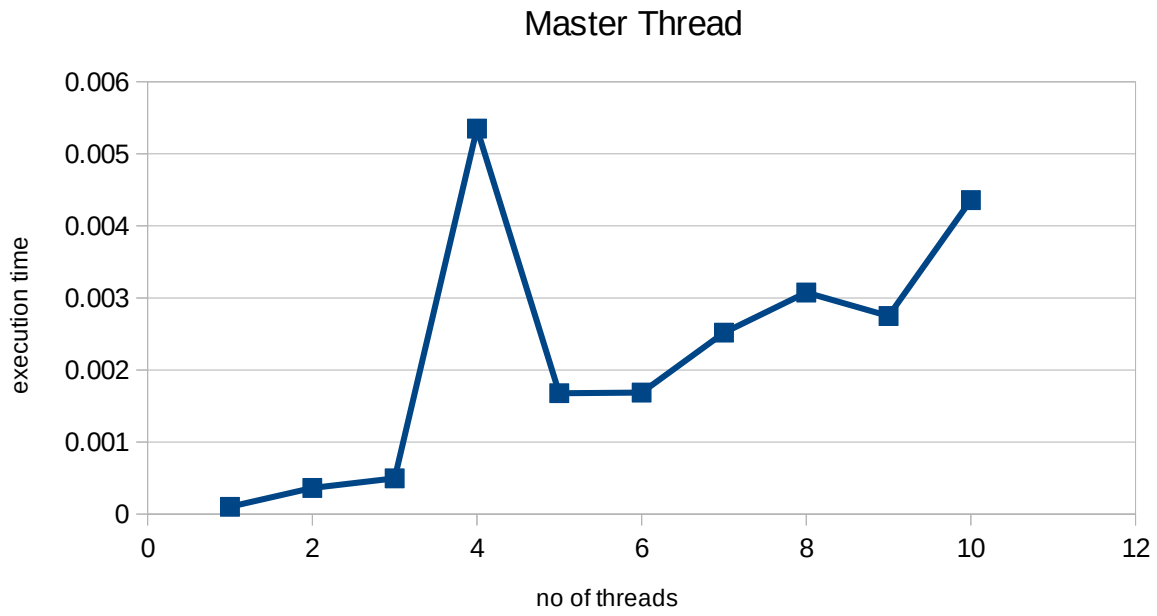
```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>

int main(int argc,char *argv[])
{
  clock_t t;
  t=clock();
  omp_set_num_threads(atoi(argv[1]));
  #pragma omp parallel
  {
    #pragma omp master
    {
      printf("Mater Thread number: %d \n",omp_get_thread_num());
    }
  }
  t=clock()-t;
```

```
  printf("Execution Time: %lfms. \n",(double)t/CLOCKS_PER_SEC);
}
```

Output:



Graph:



Result:

Hence all programs are implemented and executed successfully.

(ii) Single:

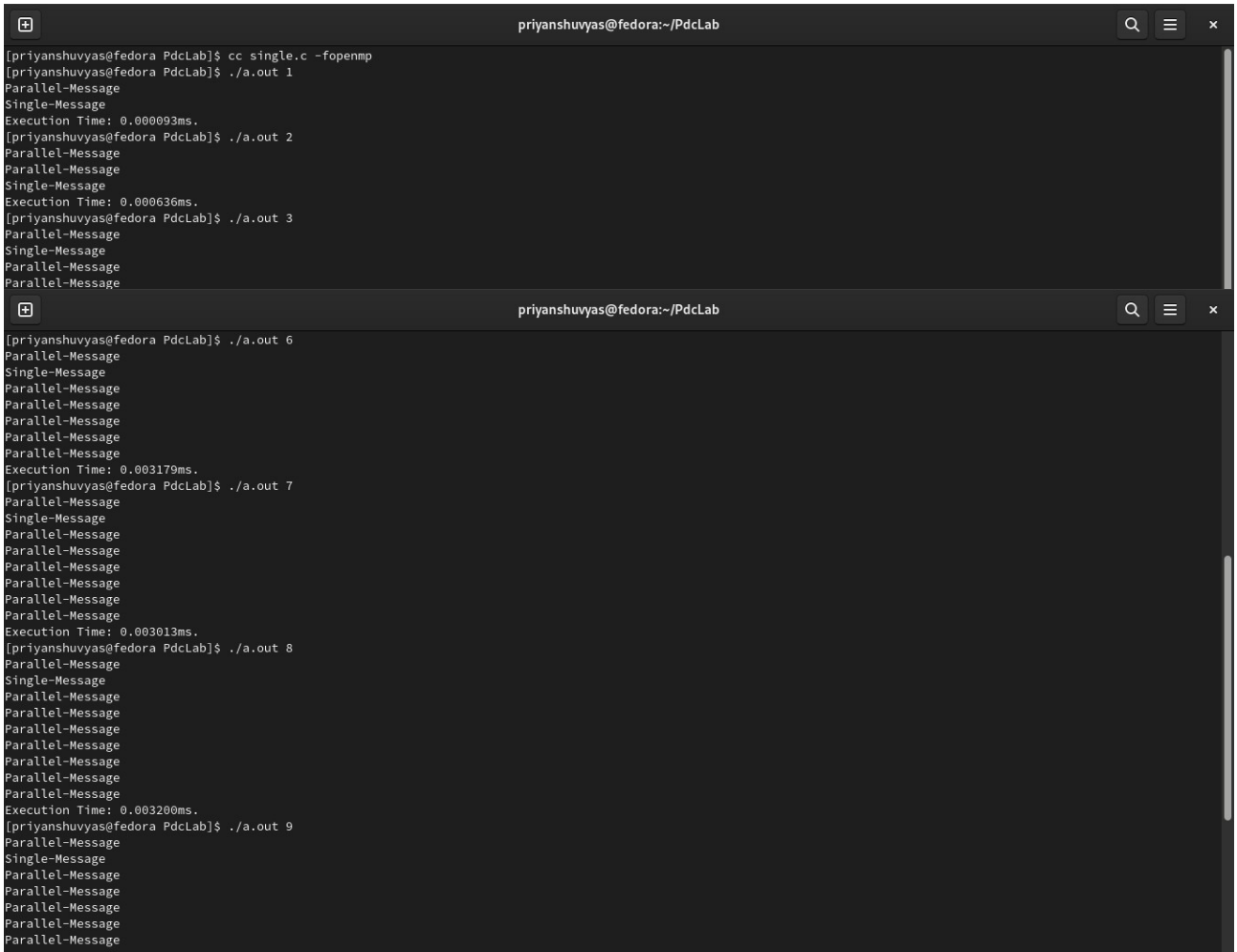
Algorithm:

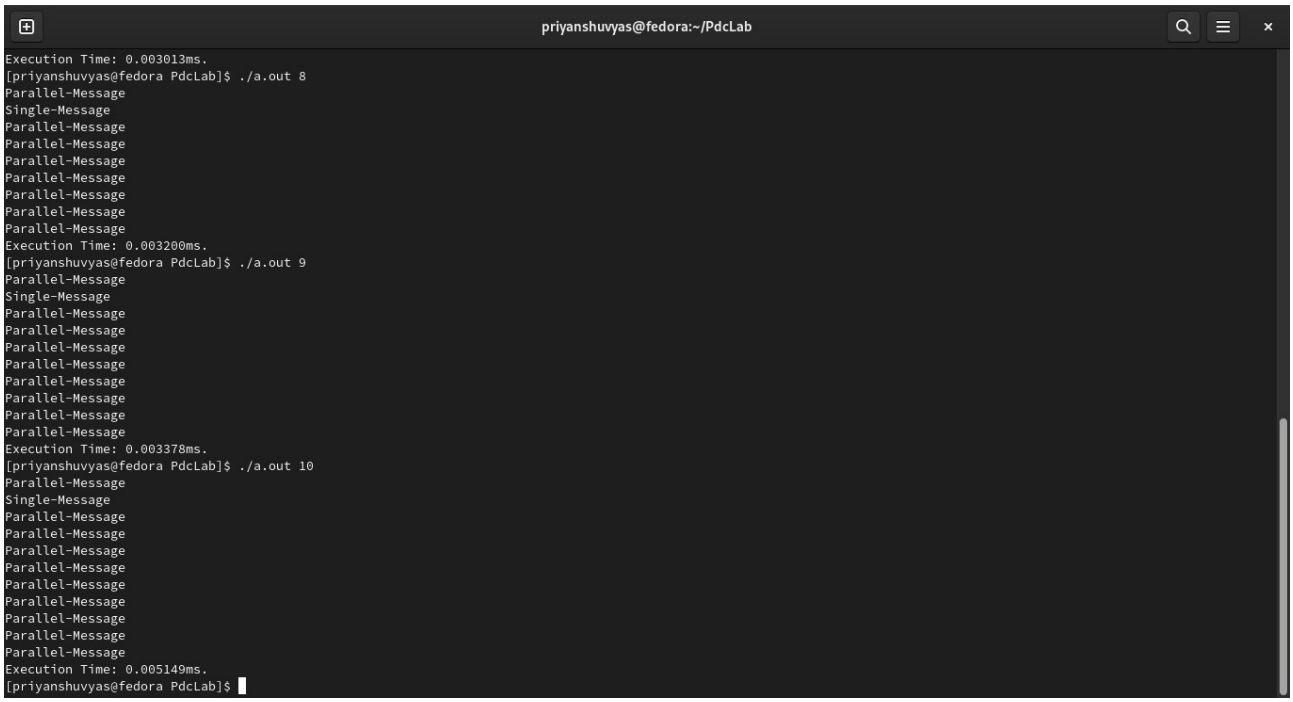
(I) set no of threads.
(II) use #pragma omp parallel.
(III) use #pragma omp single to print a statement single time.
(IV) Compute and print execution time.

Code:

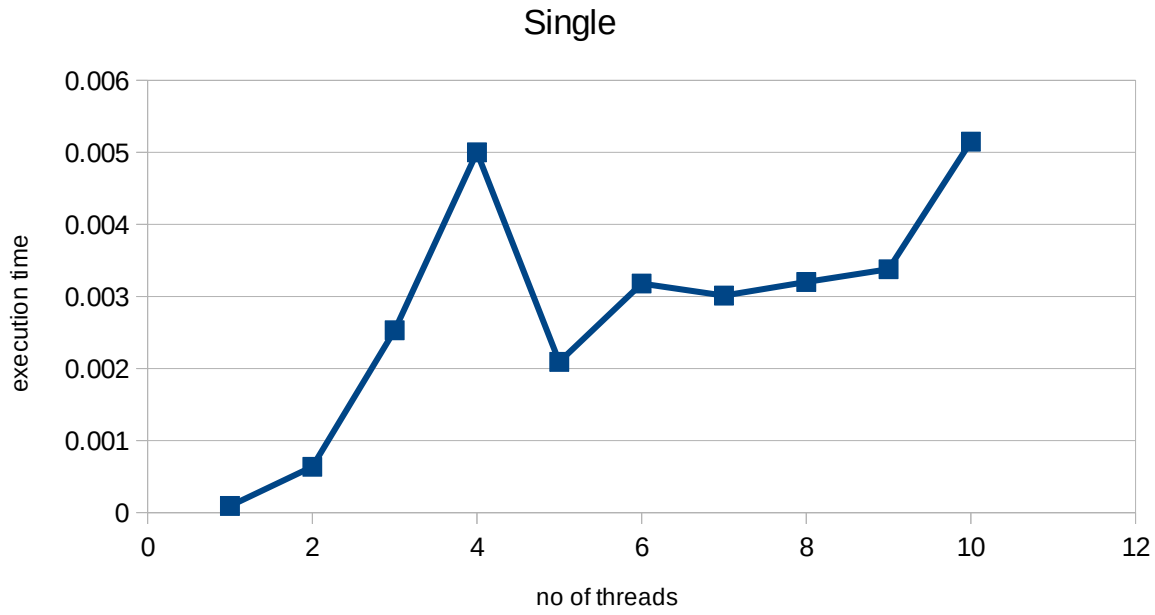
```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>

int main(int argc,char* argv[])
{
  omp_set_num_threads(atoi(argv[1]));
  clock_t t;
  t=clock();
  #pragma omp parallel
  {
    printf("Parallel-Message\n");
    #pragma omp single
    {
      printf("Single-Message\n");
    }
  }
  t=clock()-t;
  printf("Execution Time: %lfms. \n",(double)t/CLOCKS_PER_SEC);
}
```

Output:





Graph:



Result:

Hence all programs are implemented and executed successfully.

(iii) firstprivate and lastprivate

Algorithm:

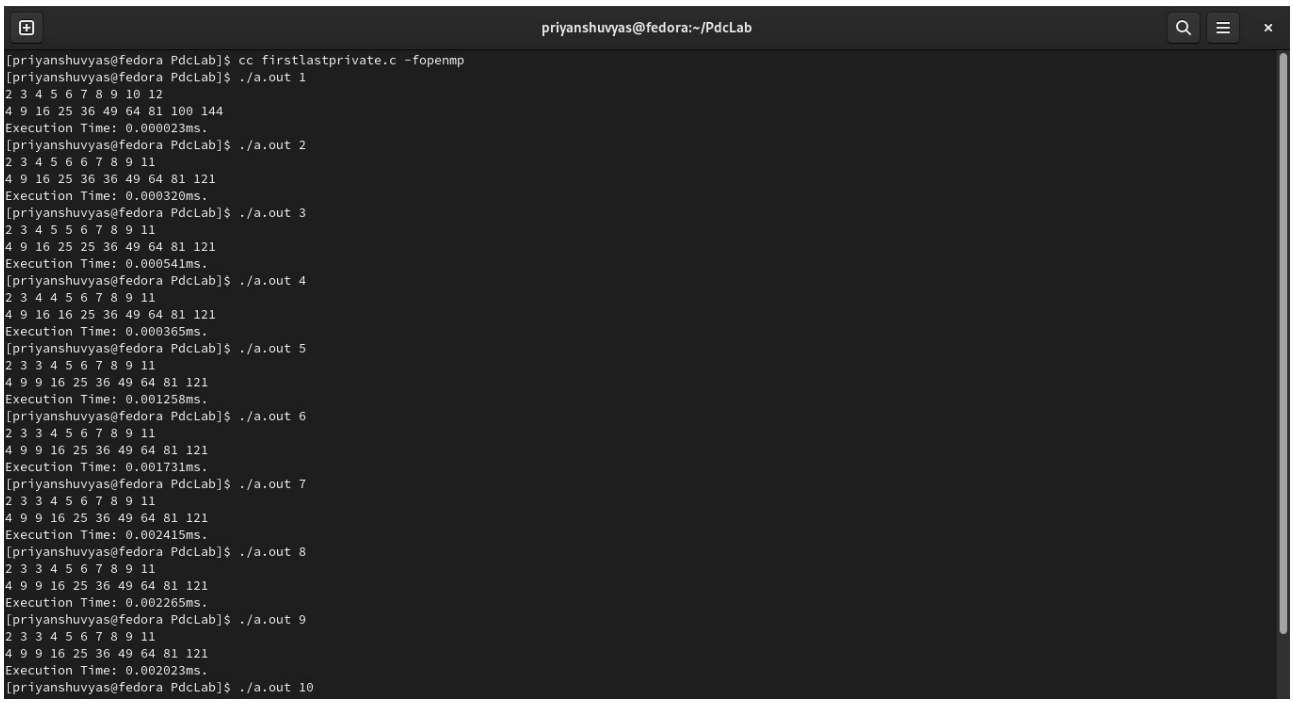
(I) set no of threads.

(II) use #pragma omp parallel for firstprivate() lastprivate().
(III) Compute and print execution time.

Code:

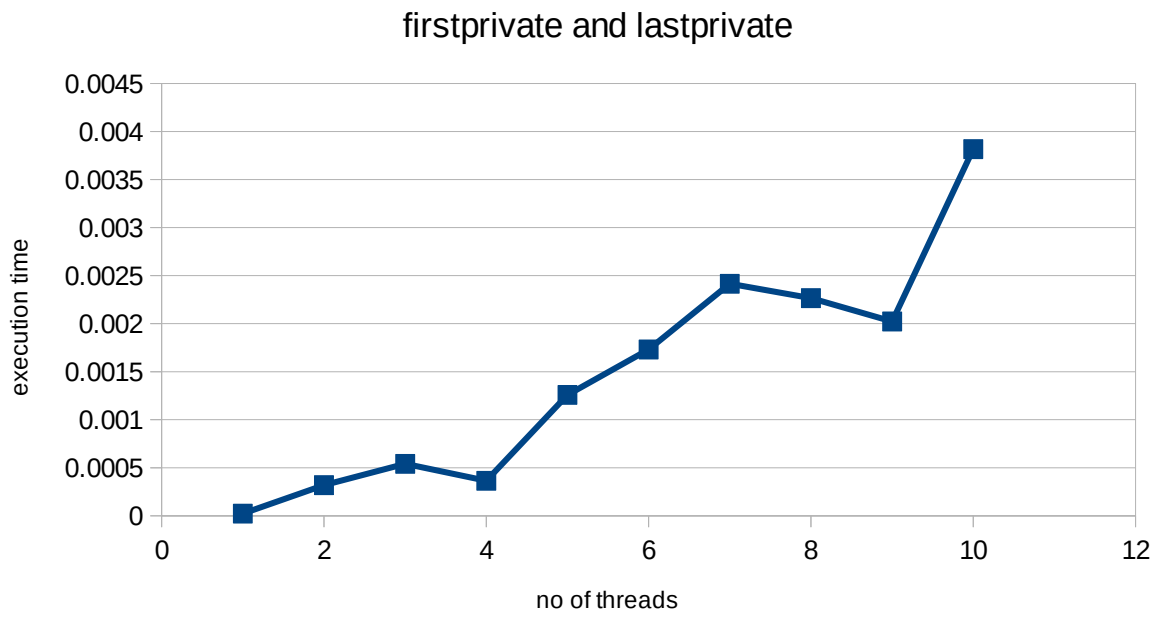
```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>

int main(int argc,char* argv[])
{
  omp_set_num_threads(atoi(argv[1]));
  clock_t t;
  t=clock();

  int j=0,x;
  int a[11]={-1,1,2,3,4,5,6,7,8,9,10};
  int b[11];
  #pragma omp parallel for firstprivate(j) lastprivate(x)
  for(int i=1;i<=10;i++)
  {
     if(i==1||i==10)
     {
       j=j+1;
     }
     a[i]=a[i]+j;
     x=a[i];
     b[i]=(x*x);
  }
  t=clock()-t;
  for(int i=1;i<=10;i++)
  {
    printf("%d ",a[i]);
  }
  printf("\n");
  for(int i=1;i<=10;i++)
  {
    printf("%d ",b[i]);
  }
  printf("\nExecution Time: %lfms. \n",(double)t/CLOCKS_PER_SEC);
}
```

Output:


priyanshuvyas@fedora:~/PdcLab
[priyanshuvyas@fedora PdcLab]$ cc firstlastprivate.c -fopenmp
[priyanshuvyas@fedora PdcLab]$ ./a.out 1
2 3 4 5 6 7 8 9 10 12
4 9 16 25 36 49 64 81 100 144
Execution Time: 0.000023ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 2
2 3 4 5 6 6 7 8 9 11
4 9 16 25 36 36 49 64 81 121
Execution Time: 0.000320ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 3
2 3 4 5 5 6 7 8 9 11
4 9 16 25 25 36 49 64 81 121
Execution Time: 0.000541ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 4
2 3 4 4 5 6 7 8 9 11
4 9 16 16 25 36 49 64 81 121
Execution Time: 0.000365ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 5
2 3 3 4 5 6 7 8 9 11
4 9 9 16 25 36 49 64 81 121
Execution Time: 0.001258ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 6
2 3 3 4 5 6 7 8 9 11
4 9 9 16 25 36 49 64 81 121
Execution Time: 0.001731ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 7
2 3 3 4 5 6 7 8 9 11
4 9 9 16 25 36 49 64 81 121
Execution Time: 0.002415ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 8
2 3 3 4 5 6 7 8 9 11
4 9 9 16 25 36 49 64 81 121
Execution Time: 0.002265ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 9
2 3 3 4 5 6 7 8 9 11
4 9 9 16 25 36 49 64 81 121
Execution Time: 0.002023ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 10


priyanshuvyas@fedora:~/PdcLab
Execution Time: 0.000023ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 2
2 3 4 5 6 6 7 8 9 11
4 9 16 25 36 36 49 64 81 121
Execution Time: 0.000320ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 3
2 3 4 5 5 6 7 8 9 11
4 9 16 25 25 36 49 64 81 121
Execution Time: 0.000541ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 4
2 3 4 4 5 6 7 8 9 11
4 9 16 16 25 36 49 64 81 121
Execution Time: 0.000365ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 5
2 3 3 4 5 6 7 8 9 11
4 9 9 16 25 36 49 64 81 121
Execution Time: 0.001258ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 6
2 3 3 4 5 6 7 8 9 11
4 9 9 16 25 36 49 64 81 121
Execution Time: 0.001731ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 7
2 3 3 4 5 6 7 8 9 11
4 9 9 16 25 36 49 64 81 121
Execution Time: 0.002415ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 8
2 3 3 4 5 6 7 8 9 11
4 9 9 16 25 36 49 64 81 121
Execution Time: 0.002265ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 9
2 3 3 4 5 6 7 8 9 11
4 9 9 16 25 36 49 64 81 121
Execution Time: 0.002023ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 10
2 2 3 4 5 6 7 8 9 11
4 4 9 16 25 36 49 64 81 121
Execution Time: 0.003816ms.
[priyanshuvyas@fedora PdcLab]$

Graph:



Result:

Hence all programs are implemented and executed successfully.

(iv) Ordered:

Algorithm:

(I) set no of threads.
(II) use #pragma omp parallel for ordered.
(III) use #pragma omp ordered to print a statement in order.
(IV) Compute and print execution time.

Code:

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

int main(int argc,char* argv[])
{
  omp_set_num_threads(atoi(argv[1]));
  clock_t t;
  t=clock();
  #pragma omp parallel for ordered
  for (int i=1;i<=10;i++)
  {
    #pragma omp ordered
    {
      printf("%d ",i);
    }
  }
```

```
  t=clock()-t;
  printf("\nExecution Time: %lfms. \n",(double)t/CLOCKS_PER_SEC);
}
```

Output:

```
priyanshuvyas@fedora:~/PdcLab
[priyanshuvyas@fedora PdcLab]$ cc ordered.c -fopenmp
[priyanshuvyas@fedora PdcLab]$ ./a.out 1
1 2 3 4 5 6 7 8 9 10
Execution Time: 0.000108ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 2
1 2 3 4 5 6 7 8 9 10
Execution Time: 0.000384ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 3
1 2 3 4 5 6 7 8 9 10
Execution Time: 0.000472ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 4
1 2 3 4 5 6 7 8 9 10
Execution Time: 0.000618ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 5
1 2 3 4 5 6 7 8 9 10
Execution Time: 0.002175ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 6
1 2 3 4 5 6 7 8 9 10
Execution Time: 0.001894ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 7
1 2 3 4 5 6 7 8 9 10
Execution Time: 0.003139ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 8
1 2 3 4 5 6 7 8 9 10
Execution Time: 0.003027ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 9
1 2 3 4 5 6 7 8 9 10
Execution Time: 0.003858ms.
[priyanshuvyas@fedora PdcLab]$ ./a.out 10
1 2 3 4 5 6 7 8 9 10
Execution Time: 0.002155ms.
[priyanshuvyas@fedora PdcLab]$
[priyanshuvyas@fedora PdcLab]$
[priyanshuvyas@fedora PdcLab]$
[priyanshuvyas@fedora PdcLab]$
[priyanshuvyas@fedora PdcLab]$
[priyanshuvyas@fedora PdcLab]$
[priyanshuvyas@fedora PdcLab]$
```

Graph:

Result:

Hence all programs are implemented and executed successfully.

Reg No. 20BCE1685
Name: Priyanshu Vyas
Course code: CSE4001
Course: Parallel and distributing computing
Title: Week 5 Lab Assignment

Q: Write an open mp program to find total number of prime numbers between 2 to N.

Algorithm:

(I) Run a for loop and use #pragma omp parallel for schedule(clause,).
(II) For all powers of 2 pass the value in function to compute total no of primes.
(III) In function noofprimes count all the prime no from 2 to N.
(IV) return count of prime numbers and print.
(V) calculate the execution time and print.

Code:

```
#include<stdio.h>
#include<omp.h>
#include<math.h>
#include<stdbool.h>
#include<time.h>
#include<stdlib.h>

bool prime(int x)
{
   int flag=0;
   for(int i=2;i<=x/2;i++)
   {
      if(x%i==0)
      {
         flag=1;
         break;
      }
   }
   return (flag==0)? true:false;
}

int noofprimes(int n)
{
   int cnt=0;
   #pragma omp parallel for reduction(+:cnt)
   for(int i=2;i<=n;i++)
   {
      if(prime(i))
      {
         cnt++;
      }
   }
   return cnt;
```

```
}

int main(int argc,char *argv[])
{
   clock_t time;
   /*int N;
   printf("Enter power of 2: ");
   scanf("%d",&N);*/
   int nthreads=atoi(argv[1]);
   omp_set_num_threads(nthreads);
   time=clock();
   printf("\n--------Static Thread Scheduling--------\n");
   #pragma omp parallel for schedule(static,3)
   for (int i=0;i<18;i++)
   {
     printf("Total no. of prime numbers between 2 to %d = %d \n",
(int)pow(2,i),noofprimes(pow(2,i)));
   }
   time=clock()-time;
   double t=(double)time/CLOCKS_PER_SEC;
   printf("Execution Time = %lfms \n",t);
   time=clock();
   printf("\n--------Dynammic Thread Scheduling--------\n");
   #pragma omp parallel for schedule(dynamic,3)
   for (int i=0;i<18;i++)
   {
     printf("Total no. of prime numbers between 2 to %d = %d \n",
(int)pow(2,i),noofprimes(pow(2,i)));
   }
   time=clock()-time;
   t=(double)time/CLOCKS_PER_SEC;
   printf("Execution Time = %lfms \n",t);
   time=clock();
   printf("\n--------Guided Thread Scheduling--------\n");
   #pragma omp parallel for schedule(guided,3)
   for (int i=0;i<18;i++)
   {
     printf("Total no. of prime numbers between 2 to %d = %d \n",
(int)pow(2,i),noofprimes(pow(2,i)));
   }
   time=clock()-time;
   t=(double)time/CLOCKS_PER_SEC;
   printf("Execution Time = %lfms \n",t);
   return 0;
}
```

Output:

For Thread 1:

```
[priyanshuvyas@fedora PdcLab]$ cc week5.c -fopenmp -lm
[priyanshuvyas@fedora PdcLab]$ ./a.out 1

--------Static Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.370684ms

--------Dynammic Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
```

```
--------Dynammic Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.366788ms

--------Guided Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.365559ms
[priyanshuvyas@fedora PdcLab]$
```

For Thread 2:

```
[priyanshuvyas@fedora PdcLab]$ cc week5.c -fopenmp -lm
[priyanshuvyas@fedora PdcLab]$ ./a.out 2

--------Static Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.437542ms

--------Dynammic Thread Scheduling--------
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
```


```
--------Dynammic Thread Scheduling--------
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
```


```
Execution Time = 2.391158ms

--------Guided Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.398254ms
[priyanshuvyas@fedora PdcLab]$
```

For Thread 3:

```
[priyanshuvyas@fedora PdcLab]$ cc week5.c -fopenmp -lm
[priyanshuvyas@fedora PdcLab]$ ./a.out 3

--------Static Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.437387ms

--------Dynammic Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
```

```
--------Dynammic Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.423852ms

--------Guided Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.458586ms
[priyanshuvyas@fedora PdcLab]$
```

For Thread 4:

```
[priyanshuvyas@fedora PdcLab]$ cc week5.c -fopenmp -lm
[priyanshuvyas@fedora PdcLab]$ ./a.out 4

--------Static Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.452797ms

--------Dynammic Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
```

```
--------Dynammic Thread Scheduling--------
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.447832ms

--------Guided Thread Scheduling--------
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.460115ms
[priyanshuvyas@fedora PdcLab]$
```

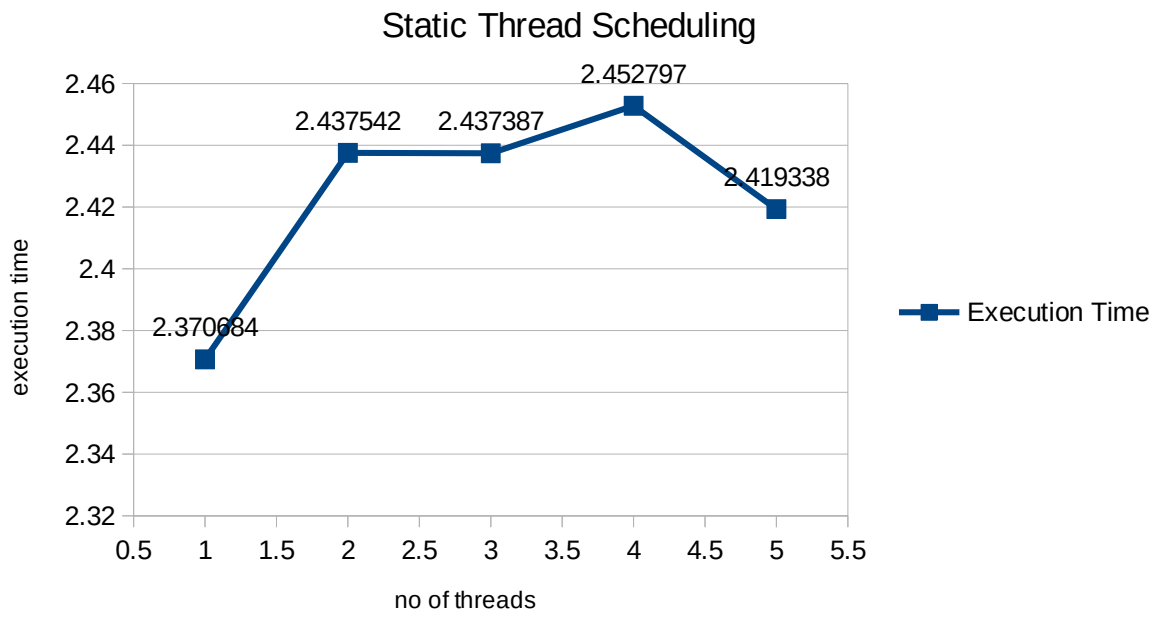
For Thread 5:

```
[priyanshuvyas@fedora PdcLab]$ cc week5.c -fopenmp -lm
[priyanshuvyas@fedora PdcLab]$ ./a.out 5

--------Static Thread Scheduling--------
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.419338ms

--------Dynammic Thread Scheduling--------
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
```

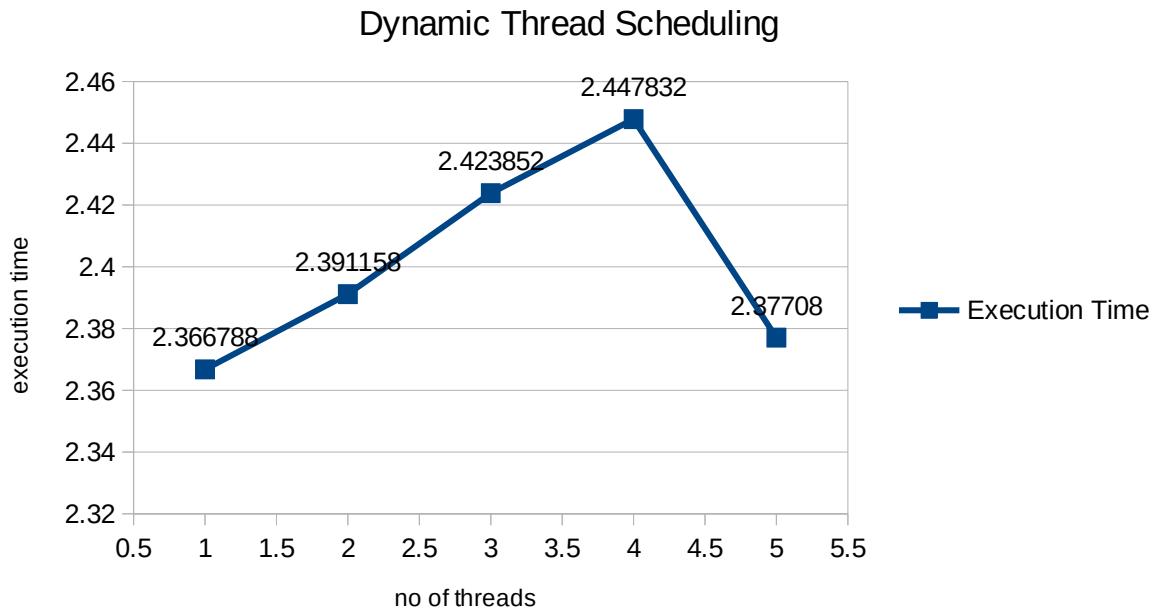
```
--------Dynammic Thread Scheduling--------
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.377080ms

--------Guided Thread Scheduling--------
Total no. of prime numbers between 2 to 16 = 6
Total no. of prime numbers between 2 to 32 = 11
Total no. of prime numbers between 2 to 64 = 18
Total no. of prime numbers between 2 to 1024 = 172
Total no. of prime numbers between 2 to 2048 = 309
Total no. of prime numbers between 2 to 1 = 0
Total no. of prime numbers between 2 to 2 = 1
Total no. of prime numbers between 2 to 4 = 2
Total no. of prime numbers between 2 to 8 = 4
Total no. of prime numbers between 2 to 128 = 31
Total no. of prime numbers between 2 to 256 = 54
Total no. of prime numbers between 2 to 512 = 97
Total no. of prime numbers between 2 to 4096 = 564
Total no. of prime numbers between 2 to 8192 = 1028
Total no. of prime numbers between 2 to 16384 = 1900
Total no. of prime numbers between 2 to 32768 = 3512
Total no. of prime numbers between 2 to 65536 = 6542
Total no. of prime numbers between 2 to 131072 = 12251
Execution Time = 2.379559ms
[priyanshuvyas@fedora PdcLab]$
```

Graph:

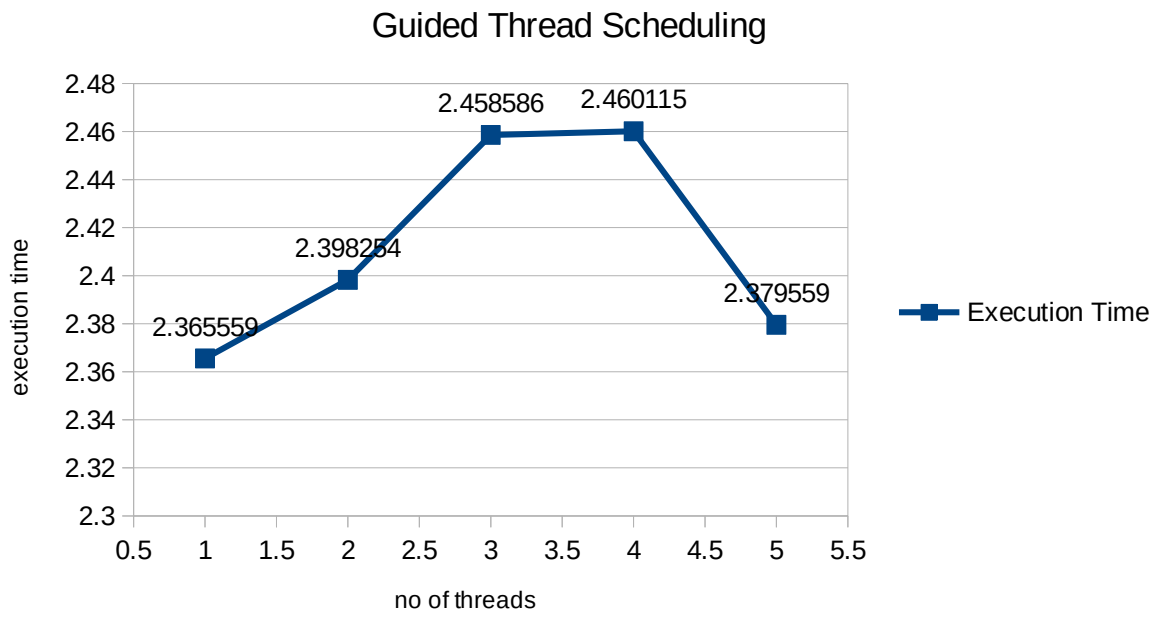
For Static Thread Scheduling:



For Dynamic Thread Scheduling:



For Guided Thread Scheduling:



Result:

Hence all programs are successfully implemented and executed.

Reg No. 20BCE1685

Name: Priyanshu Vyas

Course: CSE4001 - Parallel and Distributing Computing

Title: Week 7 – Lab Activity

Research Article 1: *Evaluating the Impact of Proposed OpenMP 5.0 Features on Performance, Portability and Productivity*

The specific contributions of this paper are as follows:

- A version of the miniMD benchmark that utilizes OpenMP 4.5 features to execute on both CPUs and GPUs, and measure its performance portability.
- Provide's an overview of features proposed in OpenMP TR7 and discuss their potential impact on developer productivity.
- Evaluate's the utility of two proposed OpenMP features – specifically metadirective and declare variant – to reduce the complexity of OpenMP 4.5 version of miniMD.

miniMD is a molecular dynamics benchmark from the Mantevo benchmark suite, acting as a proxy for LAMMPS. As a proxy, the code is missing several features found in production molecular dynamics codes: it supports only short-range force calculations using the Lennard-Jones potential or Embedded Atom Method (EAM), and supports a limited number of input configurations.

metadirective allows different directives to be used in different contexts. metadirective is similar in many ways to certain C preprocessor directives (e.g. #ifdef), but is more expressive due to its ability to query the current OpenMP context: for example, the code in Figure 1 shows how two independent instances of metadirective can work together to control whether a loop is executed in parallel on certain devices and subsequently whether updates within the loop need to be atomic.

```
// Run in parallel only when running on an offload device.
#pragma omp metadirective \
 when(device={kind(nohost)} : parallel for)
for (int i = 0; i < N; ++i)
{
  // Update atomically only when running in parallel.
  #pragma omp metadirective \
   when(construct={parallel} : atomic)
  array[i]++;
}
```

Fig. 1. An example usage of `metadirective` functionality.

One of the biggest usability gaps between OpenMP 4.5 and its alternatives lies in its reliance on functionality outside of OpenMP (e.g. pre-processors) to implement common patterns that execute different code on different devices. The metadirective and declare variant features proposed in OpenMP TR7 close this gap significantly and are sufficient to express several common idioms; however, complications arise when device-specific behaviors span multiple functions and it may be necessary to refactor existing codebases before these features can be used effectively.

Research Article 2: *Performance and energy analysis of OpenMP runtime systems with dense linear algebra algorithms*

This article analyze performance and energy consumption of five OpenMP runtime systems over a non-uniform memory access (NUMA) platform. We also selected three CPU-level optimizations or techniques to evaluate their impact on the runtime systems: processors features Turbo Boost and C-States, and CPU Dynamic Voltage and Frequency Scaling through Linux CPU Freq governors. We present an experimental study to characterize OpenMP runtime systems on the three main kernels in dense linear algebra algorithms (Cholesky, LU, and QR) in terms of performance and energy consumption. Our experimental results suggest that OpenMP runtime systems can be considered as a new energy leverage, and Turbo Boost, as well as C-States, impacted significantly performance and energy. CPU Freq governors had more impact with Turbo Boost disabled, since both optimizations reduced performance due to CPU thermal limits. An LU factorization with concurrent-write extension from libKOMP achieved up to 63% of performance gain and 29% of energy decrease over original PLASMA algorithm using GNU C compiler (GCC) lib GOMP runtime.

```
1  for (k=0; k<NB; k++) {
2  #pragma omp task untied shared(M) \
3      depend(inout: M[k*NB+k])
4    lu0(M[k*NB+k]);
5    for (j=k+1; j<NB; j++)
6  #pragma omp task untied shared(M) \
7   depend(in: M[k*NB+k]) depend(inout: M[k*NB+j])
8      fwd(M[k*NB+k], M[k*NB+j]);
9
10   for (i=k+1; i<NB; i++)
11 #pragma omp task untied shared(M)\
12  depend(in: M[k*NB+k]) depend(inout: M[i*NB+k])
13     bdiv(M[k*NB+k], M[i*NB+k]);
14
15   for (i=k+1; i<NB; i++)
16     for (j=k+1; j<NB; j++)
17 #pragma omp task untied shared(M)\
18  depend(in:M[i*NB+k], M[k*NB+j]) depend(inout:M
         [i*NB+j])
19     bmod(M[i*NB+k],M[k*NB+j],M[i*NB+j]);
20 }
```

Figure 1. LU factorization with OpenMP-dependent task.

In this article, experiments with five production-based OpenMP runtime systems and three CPU-level optimizations (Turbo Boost, C-States, and Linux CPUFreq governors) on the three main kernels in dense linear algebra were conducted on an NUMA platform. We showed that OpenMP runtime is a new leverage for controlling energy, and Turbo Boost, as well as C-States, impacted significantly performance and energy. Our experimental results suggest that small algorithmic and runtime improvements may allow performance gains up to 63% and thus reducing the energy by 29% over LU factorization from PLASMA using GCC libGOMP runtime. Runtime systems based on work stealing were more efficient in performance; although, it was not clear the impact of C-States at idle phases in order to reduce energy consumption.

Research Article 3: *Efficient OpenMP parallelization to a complex MPI parallel magnetohydrodynamics code*

The Block-Adaptive-Tree Solarwind Roe Upwind Scheme (BATS-R-US) is a multi-physics MHD code written in Fortran 90+ that has been actively developing at the University of Michigan for over 20 years. It is the most complex and often the computationally most expensive model in the Space Weather Modeling Framework (SWMF) that has been applied to simulate multi scale space physical systems including, but not limited to, the solar corona, the heliosphere, planetary magnetospheres, moons, comets and the outer heliosphere. For the purpose of adaptive mesh refinement (AMR) and running efficiency, the code was designed from the very beginning to use a 3D Cartesian block-adaptive mesh with MPI parallelization.

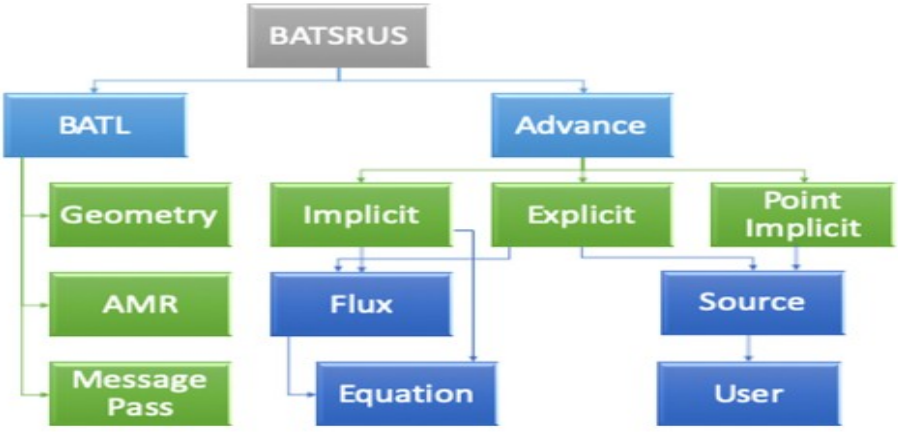


Fig. 1. The high level structure of BATS-R-US. The AMR library BATL is used for mesh generation, refinement and message passing. Explicit, fully-implicit, semi-implicit, part-implicit and point-implicit schemes can be used for time advance. Numerical flux schemes of 1st, 2nd and 5th order can be chosen. Multiple versions of the equation and user modules containing the equation variable definitions and the application specific codes are available.

In this work, we have successfully extended our finite volume/difference MHD code BATS-R-US from pure MPI to MPI + OpenMP hybrid implementation, with only 0.25% modification to the $\sim$ 250, 000 lines of source code. Good weak scaling performances are obtained up to $\sim$ 500, 000 cores with explicit time stepping and up to $\sim$ 250, 000 cores with implicit time stepping. Using the hybrid parallelization, we are now able to solve problems more than an order of magnitude larger than before thanks to the usage of shared memory for large grid arrays. We opted to use coarse-grained multi-threading applied to the loops over grid blocks, because it provides more opportunity for parallelism than the fine-grained approach applied to loops over grid cells. The main challenge with the coarse-grained approach in a large and complex code is to find and eliminate race conditions, which can be solved with well-organized code structure and the help of debuggers like Intel Inspector.

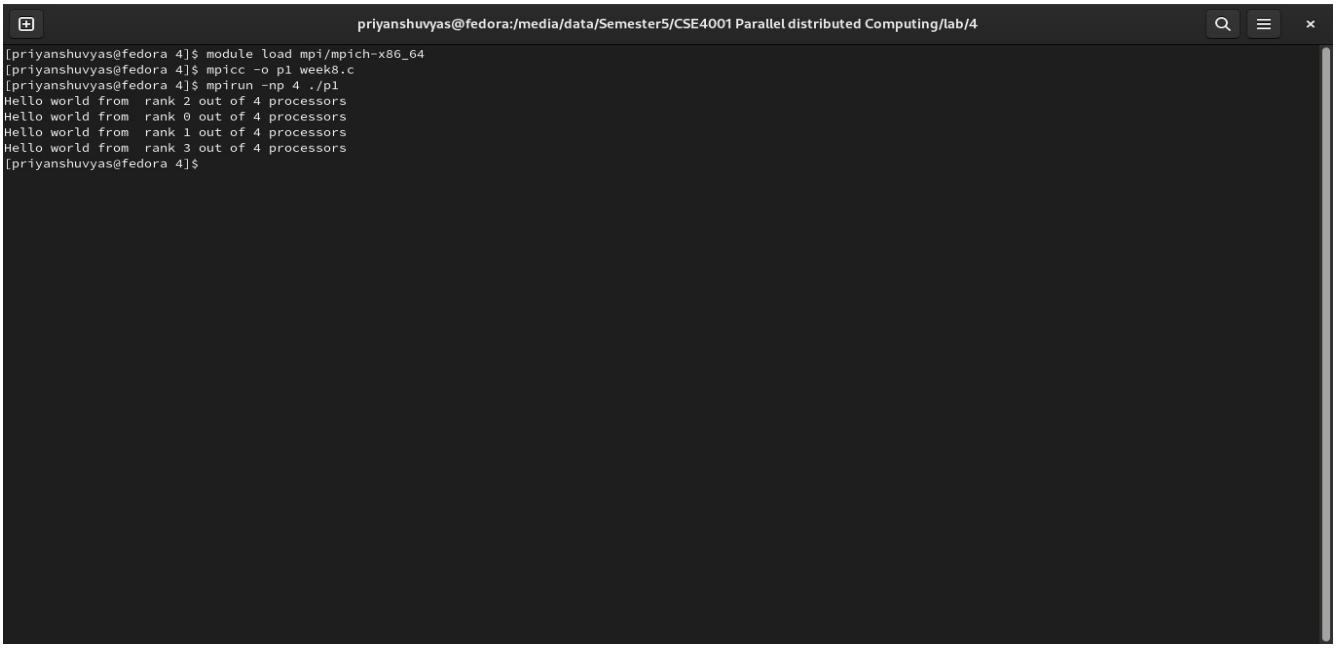
The compiler's capability to efficiently optimize a multi-threaded code varies significantly from compiler to compiler on various platforms. It is important to check the performance of the single threaded execution of the code compiled with and without OpenMP library. Hopefully all compilers will get better in optimizing hybrid parallel codes in the future.

Name: Priyanshu Vyas
Reg No. 20BCE1685
Course: CSE3002 Parallel and distributing computing
Title: Week 8 Lab Activity

Program 1 Code:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

// Print off a hello world message
    printf("Hello world from  rank %d out of %d processors\n",  world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Output:

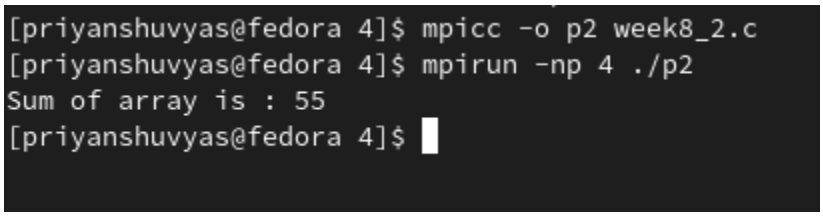


Program 2 Code:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
 // size of array
#define n 10
 int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
 // Temporary array for slave process
int a2[1000];
 int main(int argc, char* argv[])
{

    int pid, np,
        elements_per_process,
        n_elements_recieved;
    // np -> no. of processes
    // pid -> process id
    MPI_Status status;

 // Creation of parallel processes
  MPI_Init(&argc, &argv);

      // find out process ID,
    // and how many processes were started
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
// master process
    if (pid == 0) {
        int index, i;
        elements_per_process = n / np;
        // check if more than 1 processes are run
        if (np > 1) {
            // distributes the portion of array
            // to child processes to calculate
            // their partial sums
            for (i = 1; i < np - 1; i++) {
                index = i * elements_per_process;
                MPI_Send(&elements_per_process, 1, MPI_INT, i, 0,
                         MPI_COMM_WORLD);

MPI_Send(&a[index], elements_per_process,
                         MPI_INT, i, 0, MPI_COMM_WORLD);

  }
  // last process adds remaining elements
            index = i * elements_per_process;
            int elements_left = n - index;
```

```
            MPI_Send(&elements_left, 1, MPI_INT,
                  i, 0, MPI_COMM_WORLD);

            MPI_Send(&a[index], elements_left,
                  MPI_INT, i, 0, MPI_COMM_WORLD);
         }
         // master process add its own sub array
         int sum = 0;
         for (i = 0; i < elements_per_process; i++)
            sum += a[i];
            // collects partial sums from other processes
         int tmp;
         for (i = 1; i < np; i++) {
            MPI_Recv(&tmp, 1, MPI_INT, MPI_ANY_SOURCE, 0,
                  MPI_COMM_WORLD, &status);

               int sender = status.MPI_SOURCE;

            sum += tmp;
         }
           // prints the final sum of array
         printf("Sum of array is : %d\n", sum);
      }
      // slave processes
      else {
         MPI_Recv(&n_elements_recieved, 1, MPI_INT, 0, 0,
               MPI_COMM_WORLD, &status);

         // stores the received array segment
         // in local array a2
         MPI_Recv(&a2, n_elements_recieved, MPI_INT, 0, 0,
               MPI_COMM_WORLD, &status);
               // calculates its partial sum
         int partial_sum = 0;
         for (int i = 0; i < n_elements_recieved; i++)
            partial_sum += a2[i];

         // sends the partial sum to the root process
         MPI_Send(&partial_sum, 1, MPI_INT,
               0, 0, MPI_COMM_WORLD);
      }
      // cleans up all MPI state before exit of process
      MPI_Finalize();

      return 0;
}
```

Output:


[priyanshuvyas@fedora 4]$ mpicc -o p2 week8_2.c
[priyanshuvyas@fedora 4]$ mpirun -np 4 ./p2
Sum of array is : 55
[priyanshuvyas@fedora 4]$

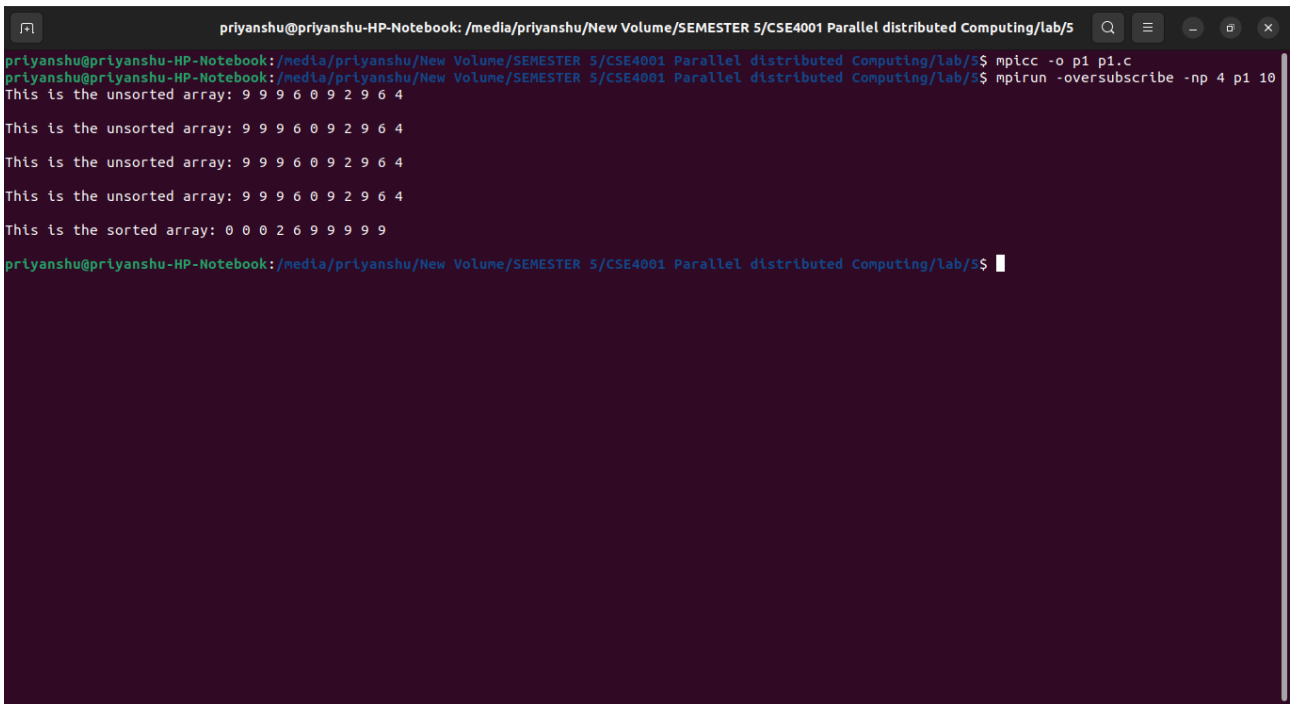
Name: Priyanshu Vyas
Reg No. 20BCE1685
Course: CSE4001 Parallel and distributing computing
Title: Week 9 Lab Activity

Program 1: Write a MPI program to sort 100 numbers.

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

void merge(int *a, int *b, int l, int m, int r) {
	int h, i, j, k;
	h = l;
	i = l;
	j = m + 1;
	while((h <= m) && (j <= r)) {
		if(a[h] <= a[j]) {
			b[i] = a[h];
			h++;
			}
		else {
			b[i] = a[j];
			j++;
			}
		i++;
		}
	if(m < h) {
		for(k = j; k <= r; k++) {
			b[i] = a[k];
			i++;
			}
		}
	else {
		for(k = h; k <= m; k++) {
			b[i] = a[k];
			i++;
			}
		}
	for(k = l; k <= r; k++) {
		a[k] = b[k];
		}
	}
void mergeSort(int *a, int *b, int l, int r) {
	int m;
	if(l < r) {
		m = (l + r)/2;
		mergeSort(a, b, l, m);
		mergeSort(a, b, (m + 1), r);
		merge(a, b, l, m, r);
		}
```

```
	}
int main(int argc, char** argv) {
	int n = atoi(argv[1]);
	int *original_array = malloc(n * sizeof(int));
	int c;
	srand(time(NULL));
	printf("This is the unsorted array: ");
	for(c = 0; c < n; c++) {
		original_array[c] = rand() % n;
		printf("%d ", original_array[c]);
		}
	printf("\n");
	printf("\n");
	int world_rank;
	int world_size;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);
	int size = n/world_size;
	int *sub_array = malloc(size * sizeof(int));
	MPI_Scatter(original_array, size, MPI_INT, sub_array, size, MPI_INT, 0,
MPI_COMM_WORLD);
	int *tmp_array = malloc(size * sizeof(int));
	mergeSort(sub_array, tmp_array, 0, (size - 1));
	int *sorted = NULL;
	if(world_rank == 0) {
		sorted = malloc(n * sizeof(int));
		}
	MPI_Gather(sub_array, size, MPI_INT, sorted, size, MPI_INT, 0, MPI_COMM_WORLD);
	if(world_rank == 0) {
		int *other_array = malloc(n * sizeof(int));
		mergeSort(sorted, other_array, 0, (n - 1));
		printf("This is the sorted array: ");
		for(c = 0; c < n; c++) {
			printf("%d ", sorted[c]);
			}
		printf("\n");
		printf("\n");
		free(sorted);
		free(other_array);
		}
	free(original_array);
	free(sub_array);
	free(tmp_array);
	MPI_Barrier(MPI_COMM_WORLD);
	MPI_Finalize();
	}
```

Output:



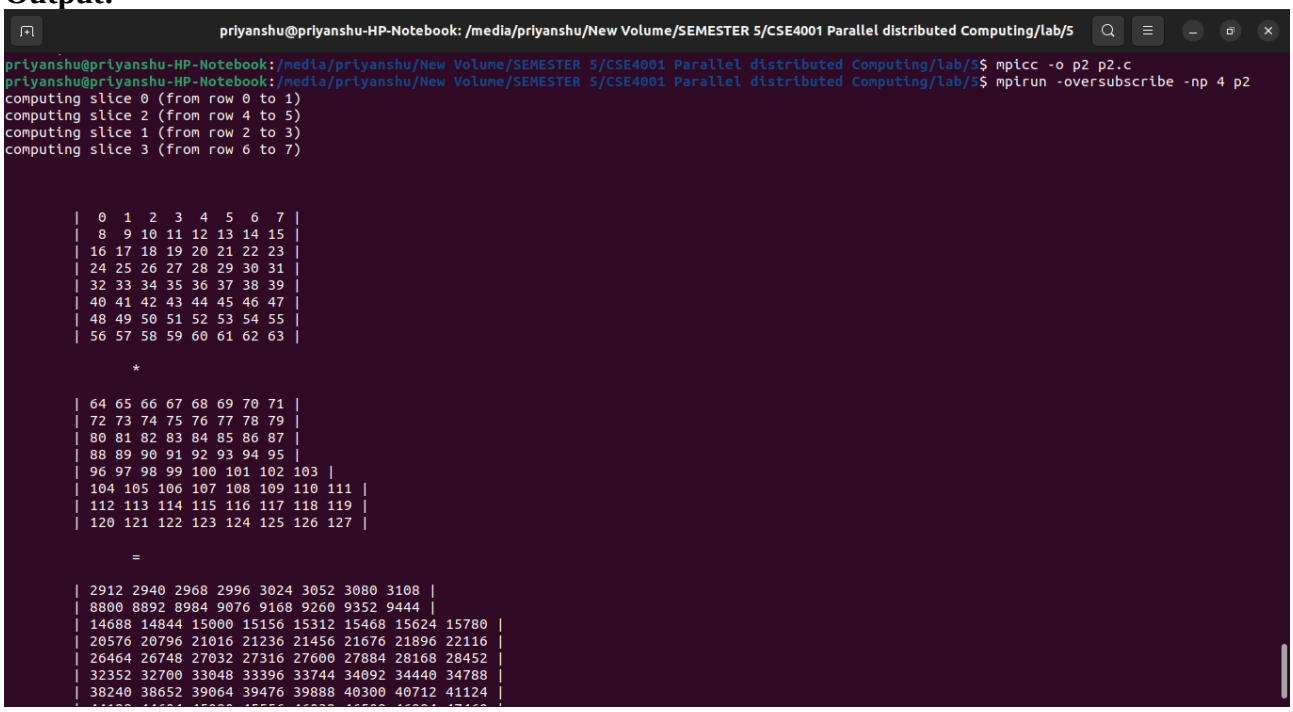
Program 2 : Write a MPI program to matrix multiplication.

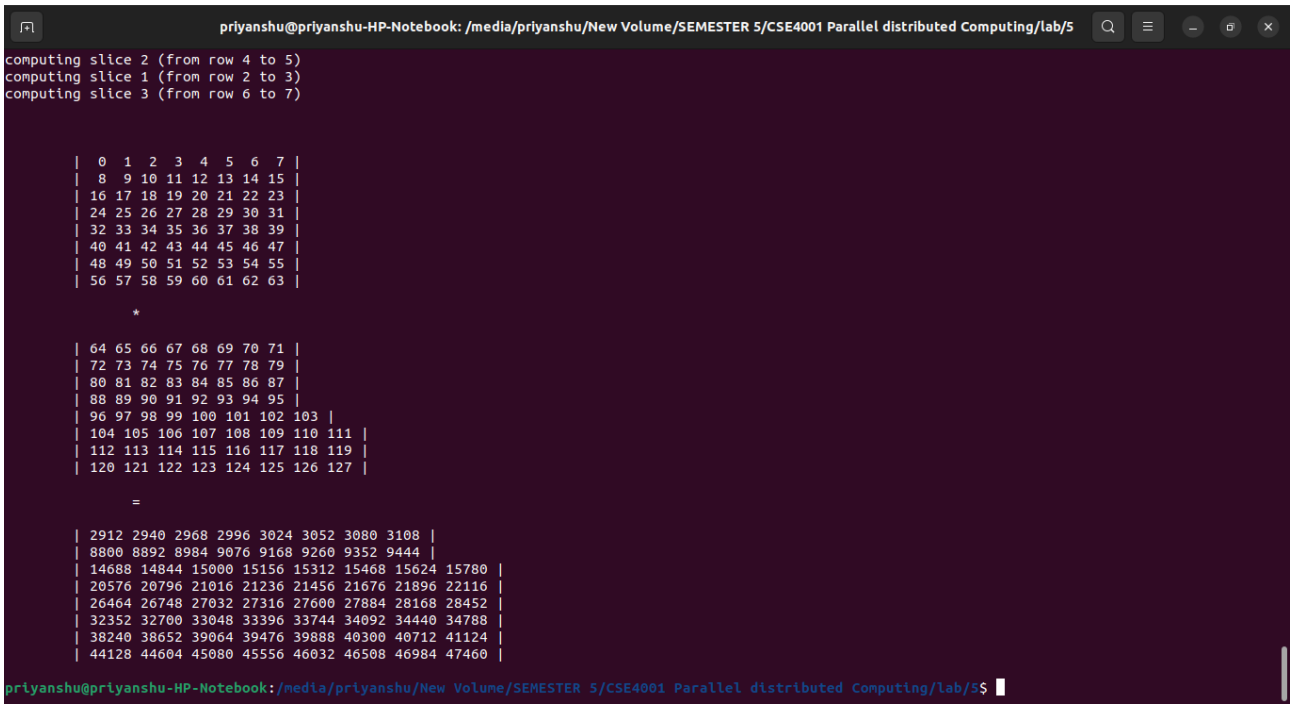
```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 8

int Matrix1[N][N], Matrix2[N][N], Result[N][N];

void fill_matrix(int m[N][N])
{
  static int n=0;
  int i, j;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      m[i][j] = n++;
}

void print_matrix(int m[N][N])
{
  int i, j = 0;
  for (i=0; i<N; i++) {
    printf("\n\t| ");
    for (j=0; j<N; j++)
      printf("%2d ", m[i][j]);
    printf("|");
  }
}
```

```
int main(int argc, char *argv[])
{
  int myrank, P, from, to, i, j, k;
  int tag = 666;            /* any value will do */
  MPI_Status status;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &P);

  if (N%P!=0) {
    if (myrank==0) printf("Matrix size not divisible by number of processors\n");
    MPI_Finalize();
    exit(-1);
  }

  from = myrank * N/P;
  to = (myrank+1) * N/P;

  if (myrank==0) {
    fill_matrix(Matrix1);
    fill_matrix(Matrix2);
  }

  MPI_Bcast (Matrix2, N*N, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Scatter (Matrix1, N*N/P, MPI_INT, Matrix1[from], N*N/P, MPI_INT, 0,
MPI_COMM_WORLD);

  printf("computing slice %d (from row %d to %d)\n", myrank, from, to-1);
  for (i=from; i<to; i++)
    for (j=0; j<N; j++) {
      Result[i][j]=0;
      for (k=0; k<N; k++)
           Result[i][j] += Matrix1[i][k]*Matrix2[k][j];
    }

  MPI_Gather (Result[from], N*N/P, MPI_INT, Result, N*N/P, MPI_INT, 0,
MPI_COMM_WORLD);

  if (myrank==0) {
    printf("\n\n");
    print_matrix(Matrix1);
    printf("\n\n\t       * \n");
    print_matrix(Matrix2);
    printf("\n\n\t       = \n");
    print_matrix(Result);
    printf("\n\n");
  }

  MPI_Finalize();
  return 0;
```

}

Output:


priyanshu@priyanshu-HP-Notebook: /media/priyanshu/New Volume/SEMESTER 5/CSE4001 Parallel distributed Computing/lab/5
priyanshu@priyanshu-HP-Notebook:/media/priyanshu/New Volume/SEMESTER 5/CSE4001 Parallel distributed Computing/lab/5$ mpicc -o p2 p2.c
priyanshu@priyanshu-HP-Notebook:/media/priyanshu/New Volume/SEMESTER 5/CSE4001 Parallel distributed Computing/lab/5$ mpirun -oversubscribe -np 4 p2
computing slice 0 (from row 0 to 1)
computing slice 2 (from row 4 to 5)
computing slice 1 (from row 2 to 3)
computing slice 3 (from row 6 to 7)
| 0 1 2 3 4 5 6 7 |
| 8 9 10 11 12 13 14 15 |
| 16 17 18 19 20 21 22 23 |
| 24 25 26 27 28 29 30 31 |
| 32 33 34 35 36 37 38 39 |
| 40 41 42 43 44 45 46 47 |
| 48 49 50 51 52 53 54 55 |
| 56 57 58 59 60 61 62 63 |
*
| 64 65 66 67 68 69 70 71 |
| 72 73 74 75 76 77 78 79 |
| 80 81 82 83 84 85 86 87 |
| 88 89 90 91 92 93 94 95 |
| 96 97 98 99 100 101 102 103 |
| 104 105 106 107 108 109 110 111 |
| 112 113 114 115 116 117 118 119 |
| 120 121 122 123 124 125 126 127 |
=
| 2912 2940 2968 2996 3024 3052 3080 3108 |
| 8800 8892 8984 9076 9168 9260 9352 9444 |
| 14688 14844 15000 15156 15312 15468 15624 15780 |
| 20576 20796 21016 21236 21456 21676 21896 22116 |
| 26464 26748 27032 27316 27600 27884 28168 28452 |
| 32352 32700 33048 33396 33744 34092 34440 34788 |
| 38240 38652 39064 39476 39888 40300 40712 41124 |


priyanshu@priyanshu-HP-Notebook: /media/priyanshu/New Volume/SEMESTER 5/CSE4001 Parallel distributed Computing/lab/5
computing slice 2 (from row 4 to 5)
computing slice 1 (from row 2 to 3)
computing slice 3 (from row 6 to 7)
| 0 1 2 3 4 5 6 7 |
| 8 9 10 11 12 13 14 15 |
| 16 17 18 19 20 21 22 23 |
| 24 25 26 27 28 29 30 31 |
| 32 33 34 35 36 37 38 39 |
| 40 41 42 43 44 45 46 47 |
| 48 49 50 51 52 53 54 55 |
| 56 57 58 59 60 61 62 63 |
*
| 64 65 66 67 68 69 70 71 |
| 72 73 74 75 76 77 78 79 |
| 80 81 82 83 84 85 86 87 |
| 88 89 90 91 92 93 94 95 |
| 96 97 98 99 100 101 102 103 |
| 104 105 106 107 108 109 110 111 |
| 112 113 114 115 116 117 118 119 |
| 120 121 122 123 124 125 126 127 |
=
| 2912 2940 2968 2996 3024 3052 3080 3108 |
| 8800 8892 8984 9076 9168 9260 9352 9444 |
| 14688 14844 15000 15156 15312 15468 15624 15780 |
| 20576 20796 21016 21236 21456 21676 21896 22116 |
| 26464 26748 27032 27316 27600 27884 28168 28452 |
| 32352 32700 33048 33396 33744 34092 34440 34788 |
| 38240 38652 39064 39476 39888 40300 40712 41124 |
| 44128 44604 45080 45556 46032 46508 46984 47460 |
priyanshu@priyanshu-HP-Notebook:/media/priyanshu/New Volume/SEMESTER 5/CSE4001 Parallel distributed Computing/lab/5$

Name: Priyanshu Vyas
Reg No.: 20BCE1685
Course: CSE4001 Parallel and distributed computing
Title: Week 10 Lab Assignment

CODE:

```
/*--------------------------------------------------
 * File:    mpi_Dijkstra.c
 *
 * Purpose: implement Dijkstra's shortest path algorithm
 *          for a weighted directed graph using mpi
 *
 * Compile: mpicc -g -Wall -o mpi_Dijkstra mpi_Dijkstra.c
 * Run:     mpiexec -n <number of processes> mpi_Dijkstra
 *
 * Input:   n: the number of vertices
 *          mat: the matrix where mat[i][j] is the length
 *               from vertex i to j
 *
 * Output:  length of the shortest path from vertex 0 to vertex v
 *          Shortest path to each vertex v from vertex 0
 *
 * Algorithm: the matrix mat is partioned by columns so that each
 *            process gets n / p columns. In each iteration each
 *            process finds its local vertex with the shortest distance
 *            from the source vertex 0. A global minimum vertex u of the found
 *            shortest distances is computed and then each process updates
 *            its local distance array if there's a shorter path that goes through u
 *
 * Note:    1. This program assumes n is evenly divisible by
 *             p (number of processes)
 *          2. Edge weights should be nonnegative
 *          3. If there is no edge between any two vertices the weight is the constant
 *             INFINITY
 *          4. The cost of traveling to itself is 0
 *          5. The adjacency matrix is stored as an 1-dimensional array and subscripts
 *             are computed using A[n * i + j] to get A[i][j] in the 2-dimensional case
 *
 *
 *--------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define INFINITY 1000000

int Read_n(int my_rank, MPI_Comm comm);
MPI_Datatype Build_blk_col_type(int n, int loc_n);
void Read_matrix(int loc_mat[], int n, int loc_n, MPI_Datatype blk_col_mpi_t,
              int my_rank, MPI_Comm comm);
void Dijkstra_Init(int loc_mat[], int loc_pred[], int loc_dist[], int loc_known[],
                int my_rank, int loc_n);
```

```
void Dijkstra(int loc_mat[], int loc_dist[], int loc_pred[], int loc_n, int n,
        MPI_Comm comm);
int Find_min_dist(int loc_dist[], int loc_known[], int loc_n);
void Print_matrix(int global_mat[], int rows, int cols);
void Print_dists(int global_dist[], int n);
void Print_paths(int global_pred[], int n);

int main(int argc, char **argv) {
   int *loc_mat, *loc_dist, *loc_pred, *global_dist = NULL, *global_pred = NULL;
   int my_rank, p, loc_n, n;
   MPI_Comm comm;
   MPI_Datatype blk_col_mpi_t;

   MPI_Init(NULL, NULL);
   comm = MPI_COMM_WORLD;
   MPI_Comm_rank(comm, &my_rank);
   MPI_Comm_size(comm, &p);
   n = Read_n(my_rank, comm);
   loc_n = n / p;
   loc_mat = malloc(n * loc_n * sizeof(int));
   loc_dist = malloc(loc_n * sizeof(int));
   loc_pred = malloc(loc_n * sizeof(int));
   blk_col_mpi_t = Build_blk_col_type(n, loc_n);

   if (my_rank == 0) {
      global_dist = malloc(n * sizeof(int));
      global_pred = malloc(n * sizeof(int));
   }
   Read_matrix(loc_mat, n, loc_n, blk_col_mpi_t, my_rank, comm);
   Dijkstra(loc_mat, loc_dist, loc_pred, loc_n, n, comm);

   /* Gather the results from Dijkstra */
   MPI_Gather(loc_dist, loc_n, MPI_INT, global_dist, loc_n, MPI_INT, 0, comm);
   MPI_Gather(loc_pred, loc_n, MPI_INT, global_pred, loc_n, MPI_INT, 0, comm);

   /* Print results */
   if (my_rank == 0) {
      Print_dists(global_dist, n);
      Print_paths(global_pred, n);
      free(global_dist);
      free(global_pred);
   }
   free(loc_mat);
   free(loc_pred);
   free(loc_dist);
   MPI_Type_free(&blk_col_mpi_t);
   MPI_Finalize();
   return 0;
}
```

```
/*---------------------------------------------------------------------
 * Function:  Read_n
 * Purpose:   Read in the number of rows in the matrix on process 0
 *            and broadcast this value to the other processes
 * In args:   my_rank:  the calling process' rank
 *            comm:  Communicator containing all calling processes
 * Ret val:   n:  the number of rows in the matrix
 */
int Read_n(int my_rank, MPI_Comm comm) {
   int n;

   if (my_rank == 0)
      scanf("%d", &n);

   MPI_Bcast(&n, 1, MPI_INT, 0, comm);
   return n;
}
```

```
/*---------------------------------------------------------------------
 * Function:  Build_blk_col_type
 * Purpose:   Build an MPI_Datatype that represents a block column of
 *            a matrix
 * In args:   n:  number of rows in the matrix and the block column
 *            loc_n = n/p:  number cols in the block column
 * Ret val:   blk_col_mpi_t:  MPI_Datatype that represents a block
 *            column
 */
MPI_Datatype Build_blk_col_type(int n, int loc_n) {
   MPI_Aint lb, extent;
   MPI_Datatype block_mpi_t;
   MPI_Datatype first_bc_mpi_t;
   MPI_Datatype blk_col_mpi_t;

   MPI_Type_contiguous(loc_n, MPI_INT, &block_mpi_t);
   MPI_Type_get_extent(block_mpi_t, &lb, &extent);

   /* MPI_Type_vector(numblocks, elts_per_block, stride, oldtype, *newtype) */
   MPI_Type_vector(n, loc_n, n, MPI_INT, &first_bc_mpi_t);

   /* This call is needed to get the right extent of the new datatype */
   MPI_Type_create_resized(first_bc_mpi_t, lb, extent, &blk_col_mpi_t);

   MPI_Type_commit(&blk_col_mpi_t);
```

```
   MPI_Type_free(&block_mpi_t);
   MPI_Type_free(&first_bc_mpi_t);

   return blk_col_mpi_t;
}

/*---------------------------------------------------------------------
 * Function:  Read_matrix
 * Purpose:   Read in an nxn matrix of ints on process 0, and
 *            distribute it among the processes so that each
 *            process gets a block column with n rows and n/p
 *            columns
 * In args:   n:  the number of rows/cols in the matrix and the submatrices
 *            loc_n = n/p:  the number of columns in the submatrices
 *            blk_col_mpi_t:  the MPI_Datatype used on process 0
 *            my_rank:  the caller's rank in comm
 *            comm:  Communicator consisting of all the processes
 * Out arg:   loc_mat:  the calling process' submatrix (needs to be
 *              allocated by the caller)
 */
void Read_matrix(int loc_mat[], int n, int loc_n,
              MPI_Datatype blk_col_mpi_t, int my_rank, MPI_Comm comm) {
   int *mat = NULL, i, j;

   if (my_rank == 0) {
      mat = malloc(n * n * sizeof(int));
      for (i = 0; i < n; i++)
         for (j = 0; j < n; j++)
            scanf("%d", &mat[i * n + j]);
   }

   MPI_Scatter(mat, 1, blk_col_mpi_t, loc_mat, n * loc_n, MPI_INT, 0, comm);

   if (my_rank == 0) free(mat);
}

/*---------------------------------------------------------------------
 * Function:   Dijkstra_Init
 * Purpose:    Initialize all the matrices so that Dijkstras shortest path
 *             can be run
 *
 * In args:    loc_n:    local number of vertices
```

```
 *            my_rank:  the process rank
 *
 * Out args:   loc_mat:  local matrix containing edge costs between vertices
 *            loc_dist: loc_dist[v] = shortest distance from the source to each vertex v
 *            loc_pred: loc_pred[v] = predecessor of v on a shortest path from source to v
 *            loc_known: loc_known[v] = 1 if vertex has been visited, 0 else
 *
 *
 */
void Dijkstra_Init(int loc_mat[], int loc_pred[], int loc_dist[], int loc_known[],
                int my_rank, int loc_n) {
   int loc_v;

   if (my_rank == 0)
      loc_known[0] = 1;
   else
      loc_known[0] = 0;

   for (loc_v = 1; loc_v < loc_n; loc_v++)
      loc_known[loc_v] = 0;

   for (loc_v = 0; loc_v < loc_n; loc_v++) {
      loc_dist[loc_v] = loc_mat[0 * loc_n + loc_v];
      loc_pred[loc_v] = 0;
   }
}




/*--------------------------------------------------------------------
 * Function:   Dijkstra
 * Purpose:    compute all the shortest paths from the source vertex 0
 *            to all vertices v
 *
 *
 * In args:    loc_mat:  local matrix containing edge costs between vertices
 *            loc_n:    local number of vertices
 *            n:        total number of vertices (globally)
 *            comm:     the communicator
 *
 * Out args:   loc_dist: loc_dist[v] = shortest distance from the source to each vertex v
 *            loc_pred: loc_pred[v] = predecessor of v on a shortest path from source to v
 *
 */
void Dijkstra(int loc_mat[], int loc_dist[], int loc_pred[], int loc_n, int n,
            MPI_Comm comm) {

   int i, loc_v, loc_u, glbl_u, new_dist, my_rank, dist_glbl_u;
   int *loc_known;
```

```
int my_min[2];
int glbl_min[2];

MPI_Comm_rank(comm, &my_rank);
loc_known = malloc(loc_n * sizeof(int));

Dijkstra_Init(loc_mat, loc_pred, loc_dist, loc_known, my_rank, loc_n);

/* Run loop n - 1 times since we already know the shortest path to global
   vertex 0 from global vertex 0 */
for (i = 0; i < n - 1; i++) {
   loc_u = Find_min_dist(loc_dist, loc_known, loc_n);

   if (loc_u != -1) {
      my_min[0] = loc_dist[loc_u];
      my_min[1] = loc_u + my_rank * loc_n;
   }
   else {
      my_min[0] = INFINITY;
      my_min[1] = -1;
   }

   /* Get the minimum distance found by the processes and store that
      distance and the global vertex in glbl_min
   */
   MPI_Allreduce(my_min, glbl_min, 1, MPI_2INT, MPI_MINLOC, comm);

   dist_glbl_u = glbl_min[0];
   glbl_u = glbl_min[1];

   /* This test is to assure that loc_known is not accessed with -1 */
   if (glbl_u == -1)
      break;

   /* Check if global u belongs to process, and if so update loc_known */
   if ((glbl_u / loc_n) == my_rank) {
      loc_u = glbl_u % loc_n;
      loc_known[loc_u] = 1;
   }

   /* For each local vertex (global vertex = loc_v + my_rank * loc_n)
      Update the distances from source vertex (0) to loc_v. If vertex
      is unmarked check if the distance from source to the global u + the
      distance from global u to local v is smaller than the distance
      from the source to local v
    */
   for (loc_v = 0; loc_v < loc_n; loc_v++) {
      if (!loc_known[loc_v]) {
         new_dist = dist_glbl_u + loc_mat[glbl_u * loc_n + loc_v];
         if (new_dist < loc_dist[loc_v]) {
            loc_dist[loc_v] = new_dist;
            loc_pred[loc_v] = glbl_u;
```

```
            }
         }
      }
   }
   free(loc_known);
}
```

```
/*---------------------------------------------------------------------
 * Function:   Find_min_dist
 * Purpose:    find the minimum local distance from the source to the
 *             assigned vertices of the process that calls the method
 *
 *
 * In args:    loc_dist:  array with distances from source 0
 *             loc_known: array with values 1 if the vertex has been visited
 *                        0 if not
 *             loc_n:     local number of vertices
 *
 * Return val: loc_u: the vertex with the smallest value in loc_dist,
 *                    -1 if all vertices are already known
 *
 * Note:       loc_u = -1 is not supposed to be used when this function returns
 *
 */
int Find_min_dist(int loc_dist[], int loc_known[], int loc_n) {
   int loc_u = -1, loc_v;
   int shortest_dist = INFINITY;

   for (loc_v = 0; loc_v < loc_n; loc_v++) {
      if (!loc_known[loc_v]) {
         if (loc_dist[loc_v] < shortest_dist) {
            shortest_dist = loc_dist[loc_v];
            loc_u = loc_v;
         }
      }
   }
   return loc_u;
}
```

```
/*---------------------------------------------------------------------
 * Function:  Print_matrix
 * Purpose:   Print the contents of the matrix
```

```
 * In args:   mat, rows, cols
 *
 *
 */
void Print_matrix(int mat[], int rows, int cols) {
   int i, j;

   for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
         if (mat[i * cols + j] == INFINITY)
            printf("i ");
         else
            printf("%d ", mat[i * cols + j]);
      printf("\n");
   }

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

```
/*--------------------------------------------------------------------
 * Function:    Print_dists
 * Purpose:     Print the length of the shortest path from 0 to each
 *              vertex
 * In args:     n:  the number of vertices
 *              dist:  distances from 0 to each vertex v:  dist[v]
 *                is the length of the shortest path 0->v
 */
void Print_dists(int global_dist[], int n) {
   int v;

   printf("  v    dist 0->v\n");
   printf("----   ---------\n");

   for (v = 1; v < n; v++) {
      if (global_dist[v] == INFINITY) {
         printf("%3d       %5s\n", v, "inf");
      }
      else
         printf("%3d       %4d\n", v, global_dist[v]);
      }
   printf("\n");
}
```

```
/*--------------------------------------------------------------------
 * Function:    Print_paths
 * Purpose:     Print the shortest path from 0 to each vertex
 * In args:     n:  the number of vertices
 *              pred:  list of predecessors:  pred[v] = u if
 *                 u precedes v on the shortest path 0->v
 */
void Print_paths(int global_pred[], int n) {
   int v, w, *path, count, i;

   path =  malloc(n * sizeof(int));

   printf("  v     Path 0->v\n");
   printf("----    ---------\n");
   for (v = 1; v < n; v++) {
      printf("%3d:    ", v);
      count = 0;
      w = v;
      while (w != 0) {
         path[count] = w;
         count++;
         w = global_pred[w];
      }
      printf("0 ");
      for (i = count-1; i >= 0; i--)
         printf("%d ", path[i]);
      printf("\n");
   }

   free(path);
}
```

OUTPUT:

```
priyanshu@priyanshu-HP-Notebook:~/Documents$ mpicc -g -Wall -o mpi_Dijkstra mpi_Dijkstra.c
priyanshu@priyanshu-HP-Notebook:~/Documents$ mpiexec -np 4 mpi_Dijkstra
Lamnodes Failed!
Check if you had booted lam before calling mpiexec else use -machinefile to pass host file to mpiexec
priyanshu@priyanshu-HP-Notebook:~/Documents$ lamboot

LAM 7.1.4/MPI 2 C++/ROMIO - Indiana University
```

```
priyanshu@priyanshu-HP-Notebook:~/Documents$ mpiexec -np 6 mpi_Dijkstra
6
0       4       2       1000000 1000000 1000000
1000000         0       5       10      1000000         1000000
1000000         1000000         0       1000000         3       1000000
1000000         1000000         1000000         0       1000000         11
1000000         1000000         1000000         4       0       1000000
1000000         1000000         1000000         1000000         1000000         0
  v    dist 0->v
----   ---------
  1          4
  2          2
  3          9
  4          5
  5         20

  v     Path 0->v
----    ---------
  1:    0 1
  2:    0 2
  3:    0 2 4 3
  4:    0 2 4
  5:    0 2 4 3 5
priyanshu@priyanshu-HP-Notebook:~/Documents$
```

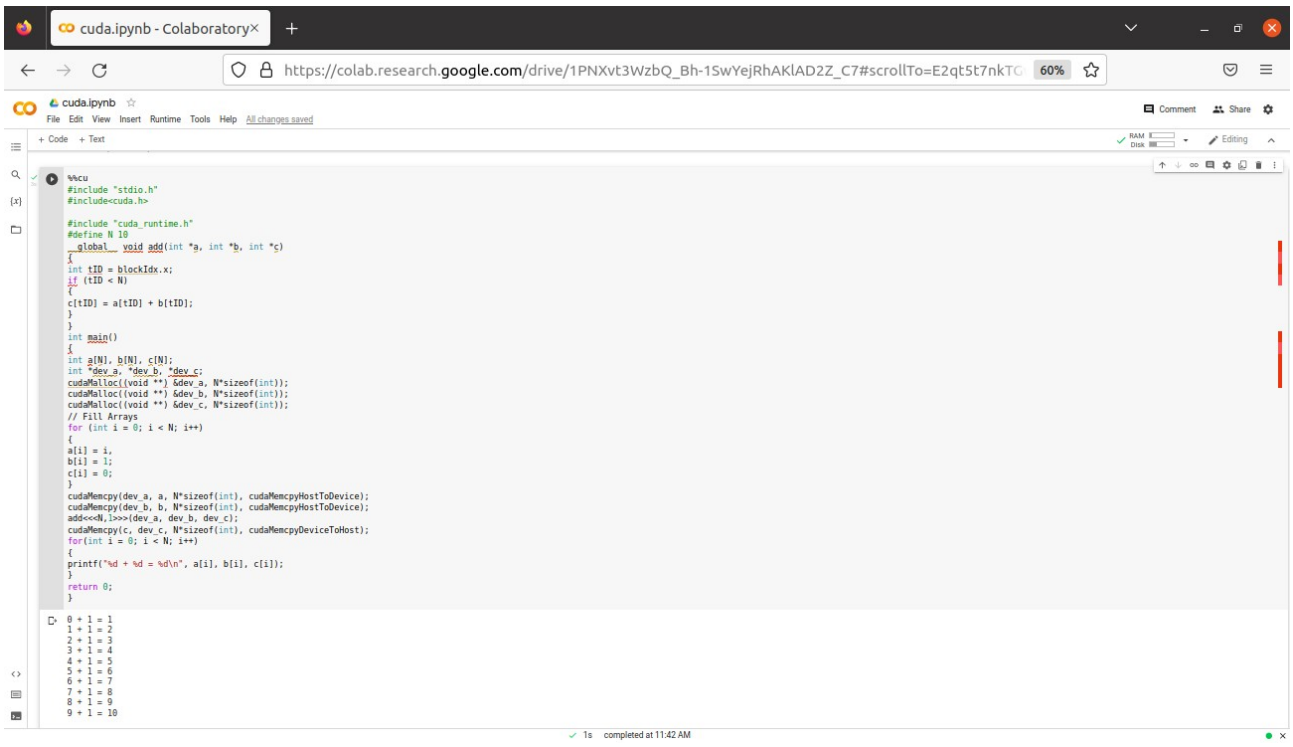
N ame: Priyanshu Vyas
Reg No. 20BCE1685
Course: CSE4001 Parallel and distributed computing
Title: Week 11 Activity Cuda Programming

Write a Cuda Program to Add two linear array.

Code:

```
#include "stdio.h"
#define N 10
__global__ void add(int *a, int *b, int *c)
{
int tID = blockIdx.x;
if (tID < N)
{
c[tID] = a[tID] + b[tID];
}
}
int main()
{
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;
cudaMalloc((void **) &dev_a, N*sizeof(int));
cudaMalloc((void **) &dev_b, N*sizeof(int));
cudaMalloc((void **) &dev_c, N*sizeof(int));
// Fill Arrays
for (int i = 0; i < N; i++)
{
a[i] = i,
b[i] = 1;
}
cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);
add<<<N,1>>>(dev_a, dev_b, dev_c);
cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
for (int i = 0; i < N; i++)
{
printf("%d + %d = %d\n", a[i], b[i], c[i]);
}
return 0;
}
```

Output:


cuda.ipynb - Colaboratory
https://colab.research.google.com/drive/1PNXvt3WzbQ_Bh-1SwYejRhAKlAD2Z_C7#scrollTo=E2qt5t7nkTC
cuda.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
Editing
%%cu
#include "stdio.h"
#include<cuda.h>
#include "cuda_runtime.h"
#define N 10
__global__ void add(int *a, int *b, int *c)
{
int tID = blockIdx.x;
if (tID < N)
{
c[tID] = a[tID] + b[tID];
}
}
int main()
{
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;
cudaMalloc((void **) &dev_a, N*sizeof(int));
cudaMalloc((void **) &dev_b, N*sizeof(int));
cudaMalloc((void **) &dev_c, N*sizeof(int));
// Fill Arrays
for (int i = 0; i < N; i++)
{
a[i] = i,
b[i] = 1;
c[i] = 0;
}
cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);
add<<<N,1>>>(dev_a, dev_b, dev_c);
cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
for(int i = 0; i < N; i++)
{
printf("%d + %d = %d\n", a[i], b[i], c[i]);
}
return 0;
}
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 1 = 4
4 + 1 = 5
5 + 1 = 6
6 + 1 = 7
7 + 1 = 8
8 + 1 = 9
9 + 1 = 10
1s completed at 11:42 AM