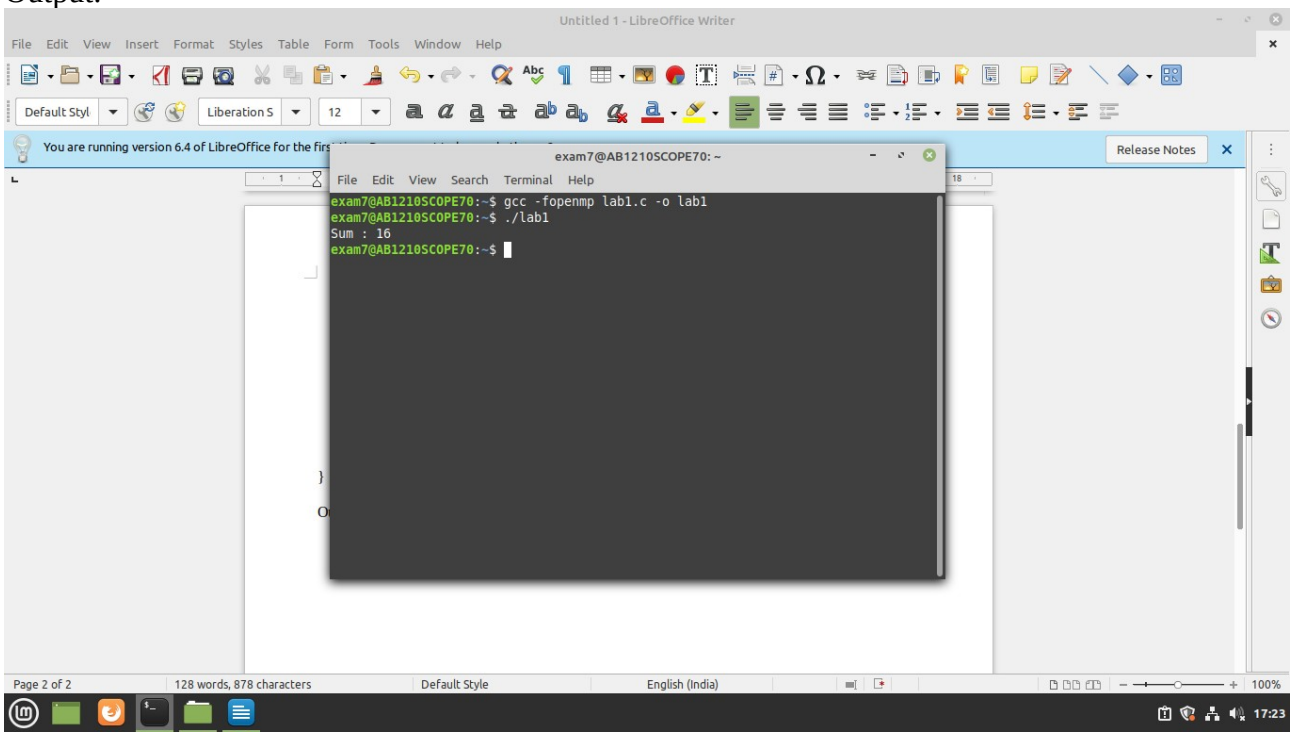
Question 1:
Code:

```
#include <omp.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>
#include <stdio.h>
#define n1 6
int main()
{
	int m=2,n=1,p=2,i;
	for(i=1;;i++)
	{
		if(i%2==0)
		{
			//printf("%d, ",m);
			m+=m;
		}
		else if(i%2!=0)
		{
			//printf("%d, ", n);
			n+=p;
			p++;
		}
		if(n>n1 && m>n1)
		break;
	}
	//printf("\n%d",i);
	int k=i;
	int arr[i];
	m=2,n=1,p=2;
	omp_set_num_threads(5);
	#pragma omp parallel for private(i) shared(arr) ordered
	for(i=1;i<=k;i++)
	{
		if(i%2==0)
		{
			//printf("%d, ",m);
			arr[i-1] = m;
			m+=m;
		}
		else if(i%2!=0)
		{
			//printf("%d, ", n);
			arr[i-1] = n;
			n+=p;
			p++;
		}
```

```
		//if(n>n1 && m>n1)
		//break;
	}
	int sum = 0;
	#pragma omp parallel for private(i) shared(arr) reduction(+: sum) ordered
	for (i = 0; i < n1; ++i) {
		sum += arr[i];
	}
	printf("Sum : %d\n",sum);
	return 0;

}
```

Output:



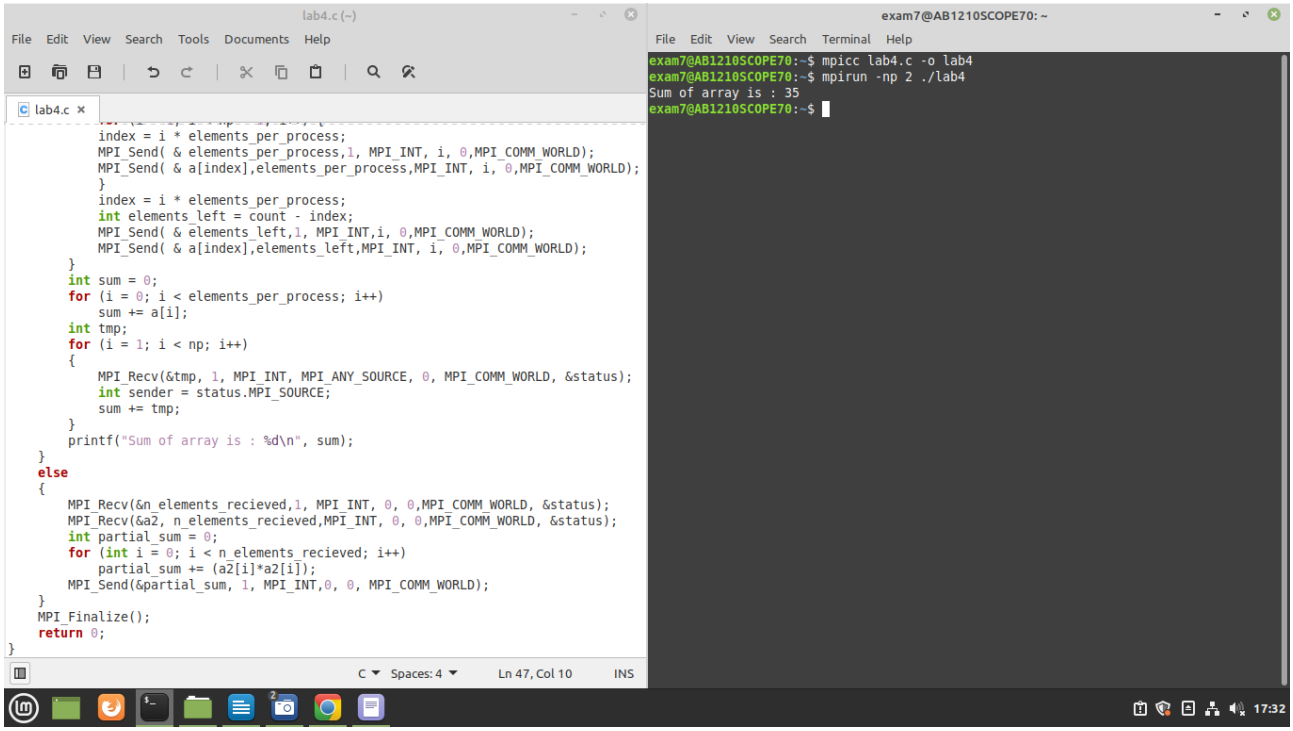

Question 2 :
Code:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define n 5
int main(int argc, char * argv[]){
	int j=0,count=0;
	for(j=1;j<=n;j++)
	{
		count++;
		j++;
	}
```

```
	int a[100],a2[100];
	int elements_per_process,elements_left, n_elements_recieved;
	int np, pid;
	int m=0;
	for(j=1;j<=n;j++)
	{
		a[m] = j;
		m++;
		j++;
	}
	MPI_Status status;
	MPI_Request request = MPI_REQUEST_NULL;
	MPI_Init( & argc, & argv);
	MPI_Comm_size(MPI_COMM_WORLD, & np);
	MPI_Comm_rank(MPI_COMM_WORLD, & pid);
	if (pid == 0) {
		int index, i;
		elements_per_process = count / np;
		if (np > 1) {
			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);
			}
			index = i * elements_per_process;
			int elements_left = count - 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);
		}
		int sum = 0;
		for (i = 0; i < elements_per_process; i++)
			sum += a[i];
		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;
		}
		printf("Sum of array is : %d\n", sum);
	}
	else
	{
		MPI_Recv(&n_elements_recieved,1, MPI_INT, 0, 0,MPI_COMM_WORLD,
&status);
		MPI_Recv(&a2, n_elements_recieved,MPI_INT, 0, 0,MPI_COMM_WORLD,
&status);
		int partial_sum = 0;
		for (int i = 0; i < n_elements_recieved; i++)
```

```
                partial_sum += (a2[i]*a2[i]);
            MPI_Send(&partial_sum, 1, MPI_INT,0, 0, MPI_COMM_WORLD);
        }
        MPI_Finalize();
        return 0;
}
```

Output:



# Lab Submission – 02


Name : Abhishek N N
Reg.No : 20BCE1025
Email : abhishek.nn2020@vitstudent.ac.in


**Program: B.Tech**

**Semester: Fall 2022-23**

**Course: CSE4001 – Parallel and Distributed Computing**

**Date: 07-08-2022**

**Exercise: 02**

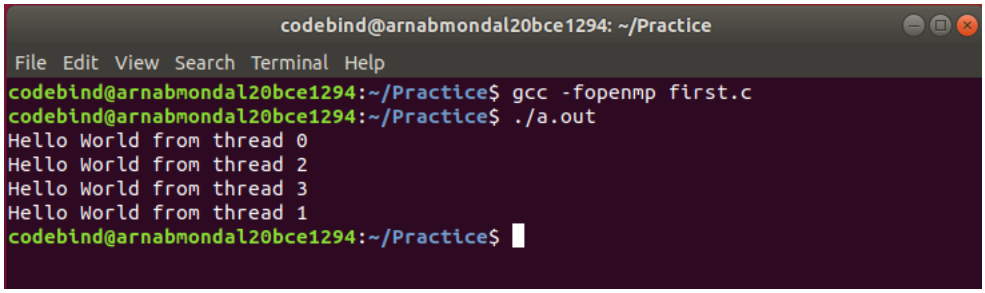
1. **Write hello world program that executes the hello world along with the thread id.**

   Aim: To make a hello world program that executes the hello world along with the thread id.

   Code:

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

int main()
{
	#pragma omp parallel
	{
		printf("Hello World from thread %d\n", omp_get_thread_num());
	}
	return 0;
}
```

   Output:




Explanation:
In the above program the "#pragma omp parallel" creates number of threads as specified. Each thread executes the print statement once. "omp_get_thread_num()" returns the thread id of the current thread in execution which is then printed in the printf() statement.

2. **Perform an OMP program to count the total number of boys and girls in each section (CSE, SW and IT) in two different campuses. Let the boys count be 3,3,3 in campus 1 and 6,7,8 in campus 2 of CSE, SW and IT. The girls count is 4, 4 ,4 in campus 1 and 5,4,3 in campus 2 of CSE, SW and IT.**

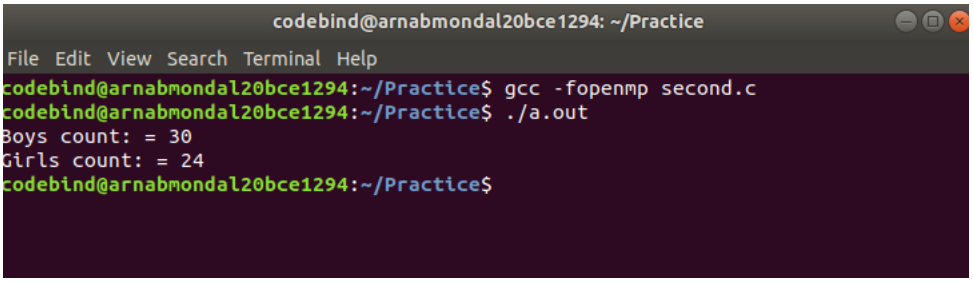
Aim:
To execute an OMP program to count the total number of boys and girls in each section (CSE, SW and IT) in two different campuses. Let the boys count be 3,3,3 in campus 1 and 6,7,8 in campus 2 of CSE, SW and IT. The girls count is 4, 4 ,4 in campus 1 and 5,4,3 in campus 2 of CSE, SW and IT.

Code:

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

int main()
{
        int n1[]={3,3,3};
        int n2[]={6,7,8};
        int n3[] = {4,4,4};
        int n4[] = {5,4,3};
        int boys = 0;
        int girls = 0;
        #pragma omp parallel num_threads(3)
        {
                int id = omp_get_thread_num();
                boys += n1[id] + n2[id];
                girls += n3[id] + n4[id];
        }
        printf("Boys count: = %d\n", boys);
```

```
        printf("Girls count: = %d\n",girls);
    return 0;
}
```

Output:




Explanation:
In the above code "#pragma omp parallel num_threads(3)" creates three threads to be executed in parallel. In the body we store the value of thread if in a variable returned by the function "omp_get_thread_num()". We then add the value of two arrays at index number equal to the thread id returned. Since there are three threads created, thread ids would 0,1,2 which corresponds to the index number of all the elements in an array and hence with each thread execution each element of an array is visited and added to find the total sum.

3. **Let there be two vectors [3, 2, -1] and [5, -4, 3]. Find the dot product of the vectors ($\vec{a}\cdot\vec{b}$ = a1a2 + b1b2 + c1c2).**

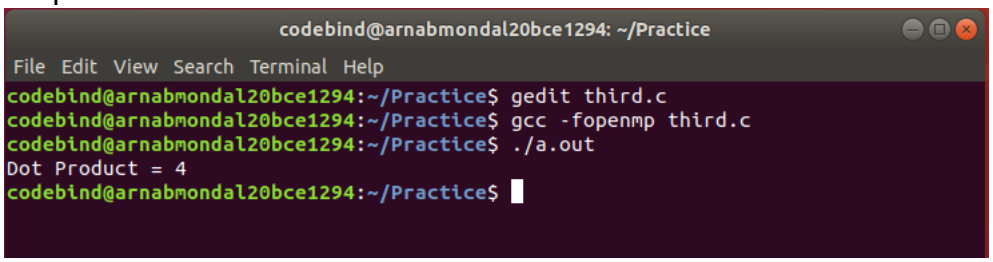
Aim:
To find dot product of two given vectors.

Code:

```
#include <omp.h>

#include <stdio.h>



int main()

{

        int n1[]={3,2,-1};

        int n2[]={5,-4,3};

        int dp=0;
```

```
        #pragma omp parallel num_threads(3)
        {
                int id = omp_get_thread_num();
                dp += n1[id] * n2[id];
        }
        printf("Dot Product = %d\n", dp);
}
```

Output:




Explanation:

In the above code “#pragma omp parallel num_threads(3)” creates three threads to be executed in parallel. In the body we store the value of thread if in a variable returned by the function “omp_get_thread_num()”. We then multiply the value of two arrays at index number equal to the thread id returned. Since there are three threads created, thread ids would 0,1,2 which corresponds to the index number of all the elements in an array and hence with each thread execution each element of an array is visited, multiplied and added to find the dot product.

# Lab Submission – 03


Name : Abhishek N N
Reg.No : 20BCE1025
Email : abhishek.nn2020@vitstudent.ac.in


**Program: B.Tech**

**Semester: Fall 2022-23**

**Course: CSE4001 – Parallel and Distributed Computing**

**Date: 22-08-2022**

**Exercise: 03**

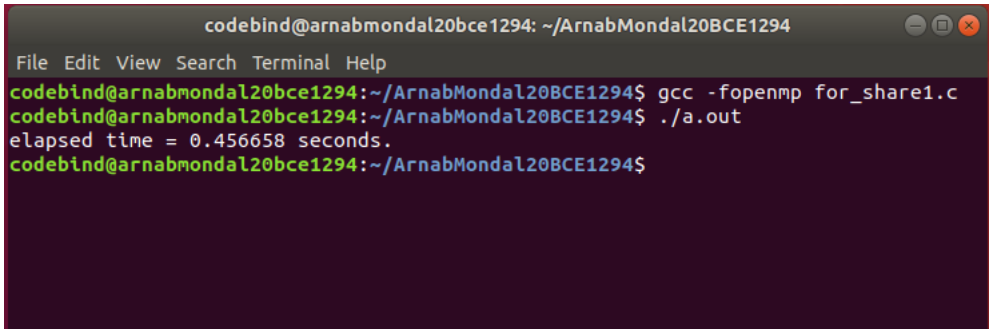
**Number of threads : 1**

**Matrix Size : 500**

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>


#define N 500

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i,j,k;
    double elapsed, start, end;
    omp_set_num_threads(1);
    for (i= 0; i< N; i++)
        for (j= 0; j< N; j++)
```

```
    {
            A[i][j] = 2;
            B[i][j] = 2;
    }

    start = omp_get_wtime();
    #pragma omp parallel for private(i,j,k) shared(A,B,C)
    //schedule(static)
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            for (k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }


    end = omp_get_wtime();

    elapsed = end - start;
    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:




**Number of threads : 1**

**Matrix Size : 1000**

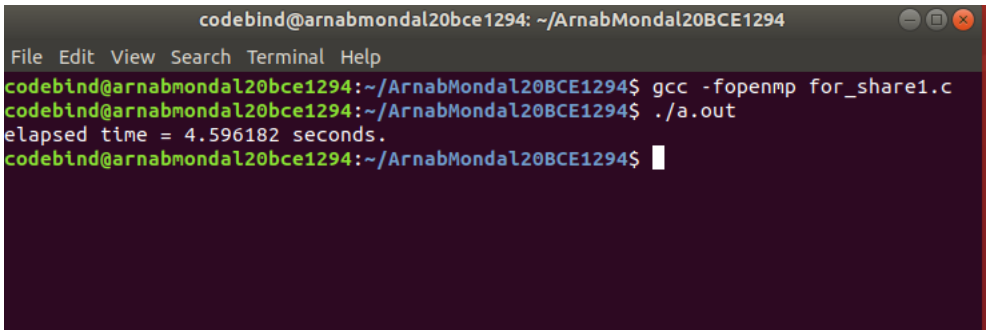
Code:

```
#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>
```

```c
#include <sys/time.h>

#define N 1000

int A[N][N];

int B[N][N];

int C[N][N];

int main()

{

    int i, j, k;

    double elapsed, start, end;

    omp_set_num_threads(1);

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

        for (j = 0; j < N; j++)

        {

            A[i][j] = 2;

            B[i][j] = 2;
        }

    start = omp_get_wtime();

#pragma omp parallel for private(i, j, k) shared(A, B, C)

    // schedule(static)

    for (i = 0; i < N; ++i)
    {

        for (j = 0; j < N; ++j)
        {

            for (k = 0; k < N; ++k)
            {

                C[i][j] += A[i][k] * B[k][j];
            }
```

```
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;

    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:




**Number of threads : 4**

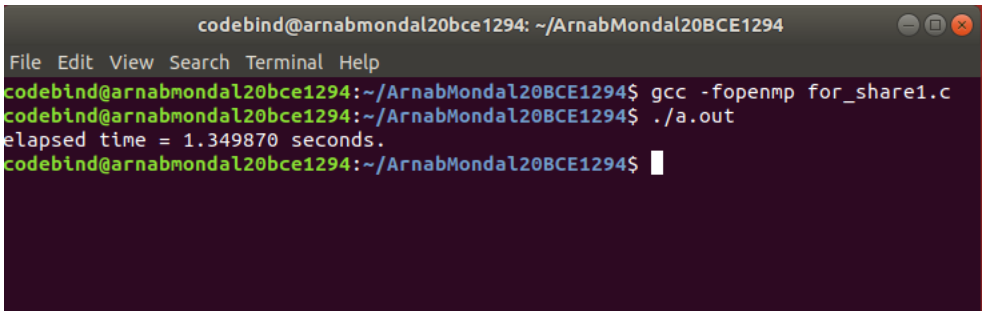
**Matrix Size : 1000**

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>

#define N 1000

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i, j, k;
    double elapsed, start, end;
    omp_set_num_threads(4);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
```

```
            A[i][j] = 2;
            B[i][j] = 2;
        }

    start = omp_get_wtime();
#pragma omp parallel for private(i, j, k) shared(A, B, C)
    // schedule(static)
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            for (k = 0; k < N; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;
    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:




**Number of threads : 5**

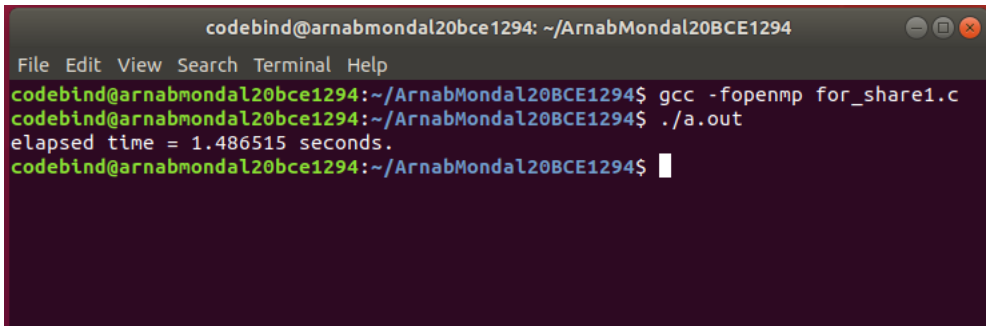
**Matrix Size : 1000**

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>
```

```c
#define N 1000

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i, j, k;
    double elapsed, start, end;
    omp_set_num_threads(5);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            A[i][j] = 2;
            B[i][j] = 2;
        }

    start = omp_get_wtime();
#pragma omp parallel for private(i, j, k) shared(A, B, C)
    // schedule(static)
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            for (k = 0; k < N; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;
    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:



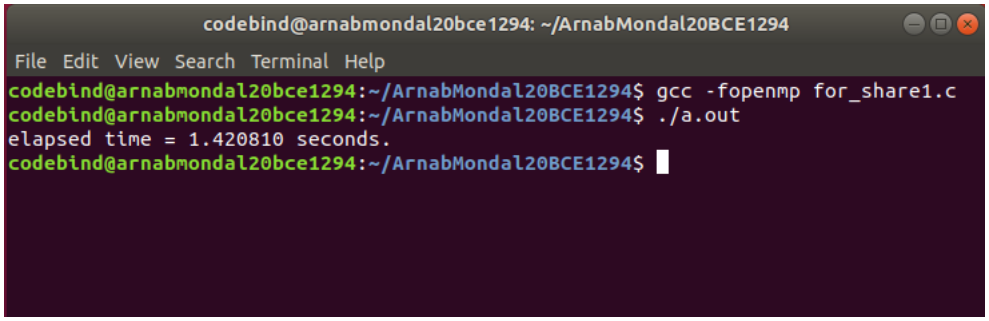
**Number of threads : 6**

**Matrix Size : 1000**

Code:

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>

#define N 1000

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i, j, k;
    double elapsed, start, end;
    omp_set_num_threads(6);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            A[i][j] = 2;
            B[i][j] = 2;
        }

    start = omp_get_wtime();
#pragma omp parallel for private(i, j, k) shared(A, B, C)
    // schedule(static)
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            for (k = 0; k < N; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;
    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:




**Number of threads : 7**

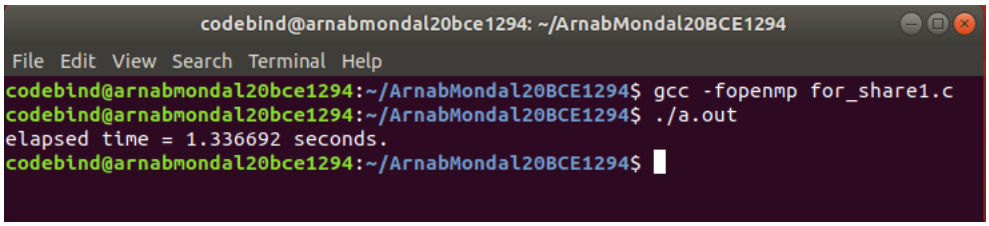
**Matrix Size : 1000**

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>

#define N 1000

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i, j, k;
    double elapsed, start, end;
    omp_set_num_threads(7);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            A[i][j] = 2;
            B[i][j] = 2;
        }

    start = omp_get_wtime();
#pragma omp parallel for private(i, j, k) shared(A, B, C)
    // schedule(static)
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
```

```
        {
            for (k = 0; k < N; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;
    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:




**Number of threads : 8**

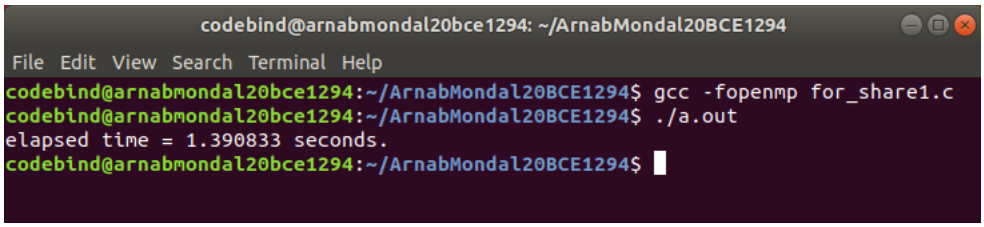
**Matrix Size : 1000**

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>

#define N 1000

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i, j, k;
    double elapsed, start, end;
    omp_set_num_threads(8);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
```

```
        {
            A[i][j] = 2;
            B[i][j] = 2;
        }

    start = omp_get_wtime();
#pragma omp parallel for private(i, j, k) shared(A, B, C)
    // schedule(static)
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            for (k = 0; k < N; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;
    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:




Speedup for 4 threads – $\frac{4.596182}{1.349870}$ = 3.404907

Speedup for 5 threads – $\frac{4.596182}{1.486515}$ = 3.091917

Speedup for 6 threads – $\frac{4.596182}{1.420810}$ = 3.234902

Speedup for 7 threads – $\frac{4.596182}{1.336692}$ = 3.438474

Speedup for 8 threads – $\frac{4.596182}{1.390833}$ = 3.304625

Efficiency for 4 threads – $\frac{3.404907}{4}$ = 0.851226 = 85.12%

Efficiency for 5 threads – $\frac{3.091917}{5}$ = 0.618383 = 61.83%

Efficiency for 6 threads – $\frac{3.234902}{6}$ = 0.539150 = 53.91%

Efficiency for 7 threads – $\frac{3.438474}{7}$ = 0.491210 = 49.12%

Efficiency for 8 threads – $\frac{3.304625}{8}$ = 0.413078 = 41.30%

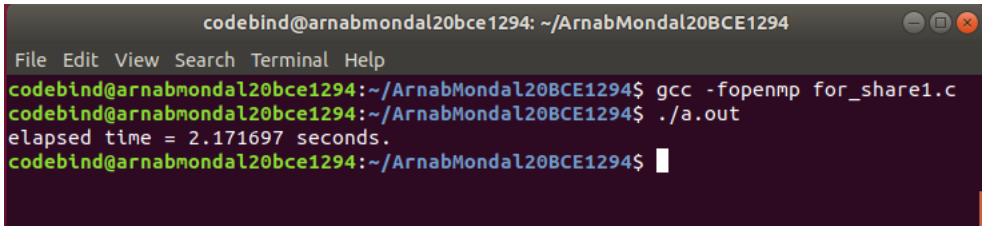
**(use loop sharing, matrix multiplication,1000,8,static)**

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>
#define N 1000
int A[N][N];
int B[N][N];
int C[N][N];
int main()
{
    int i, j, k;
    double elapsed, start, end;
    omp_set_num_threads(8);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
```

```
            A[i][j] = 2;

            B[i][j] = 2;
        }

    start = omp_get_wtime();

#pragma omp parallel for private(i, j, k) shared(A, B, C) schedule(static)

    for (i = 0; i < N; ++i)
    {

        for (j = 0; j < N; ++j)
        {

            for (k = 0; k < N; ++k)
            {

                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;

    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:




**(use loop sharing, matrix multiplication,500,8,dynamic)**

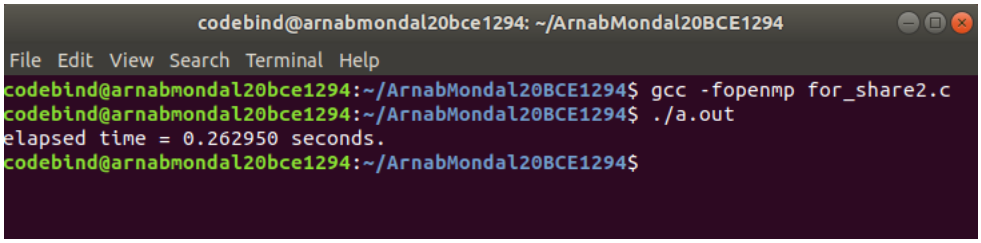
Code:

```
#include <pthread.h>

#include <stdio.h>
```

```c
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>
#define N 500
int A[N][N];
int B[N][N];
int C[N][N];
int main()
{
    int i, j, k;
    double elapsed, start, end;
    omp_set_num_threads(8);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            A[i][j] = 2;
            B[i][j] = 2;
        }
    start = omp_get_wtime();
#pragma omp parallel for private(i, j, k) shared(A, B, C) schedule(dynamic)
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            for (k = 0; k < N; ++k)
            {
```

```
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;

    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:



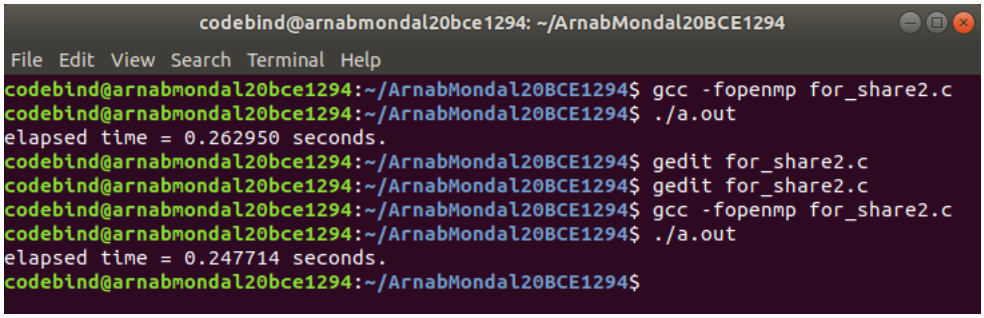

**(use loop sharing, matrix multiplication,500,8,dynamic,ordered)**

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>

#define N 500

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i, j, k;
    double elapsed, start, end;
    omp_set_num_threads(8);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            A[i][j] = 2;
            B[i][j] = 2;
```

```
        }

    start = omp_get_wtime();
#pragma omp parallel for private(i, j, k) shared(A, B, C) schedule(dynamic)
ordered
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            for (k = 0; k < N; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;
    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:



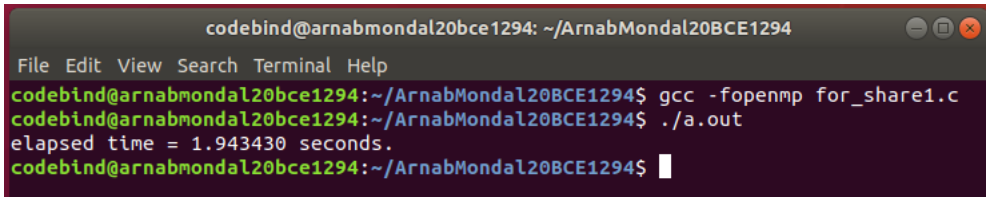

**(use loop sharing, matrix multiplication,1000,8,dynamic,ordered)**

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>

#define N 1000

int A[N][N];
int B[N][N];
```

```
int C[N][N];

int main()
{
    int i, j, k;
    double elapsed, start, end;
    omp_set_num_threads(8);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            A[i][j] = 2;
            B[i][j] = 2;
        }

    start = omp_get_wtime();
#pragma omp parallel for private(i, j, k) shared(A, B, C) schedule(static)
ordered
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            for (k = 0; k < N; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;
    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:



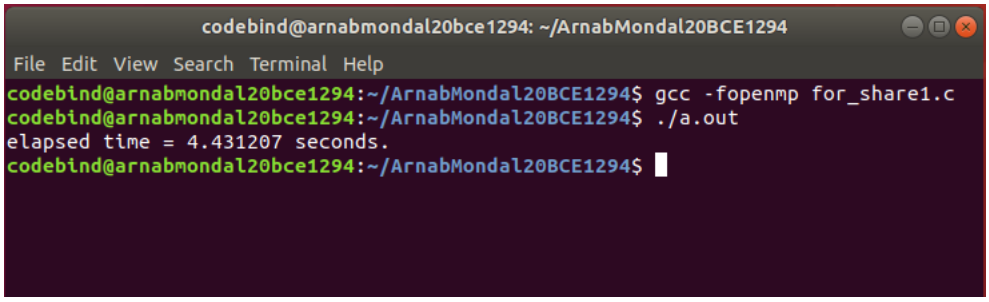

## Using nowait and static

Code:

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

```c
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#include <sys/time.h>

#define N 1000

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i, j, k;
    double elapsed, start, end;
    omp_set_num_threads(8);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            A[i][j] = 2;
            B[i][j] = 2;
        }

    start = omp_get_wtime();
//#pragma omp parallel for private(i,j,k) shared(A,B,C) schedule(static)
ordered nowait
#pragma omp for private(i, j, k) schedule(static) ordered nowait
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            for (k = 0; k < N; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;
    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:



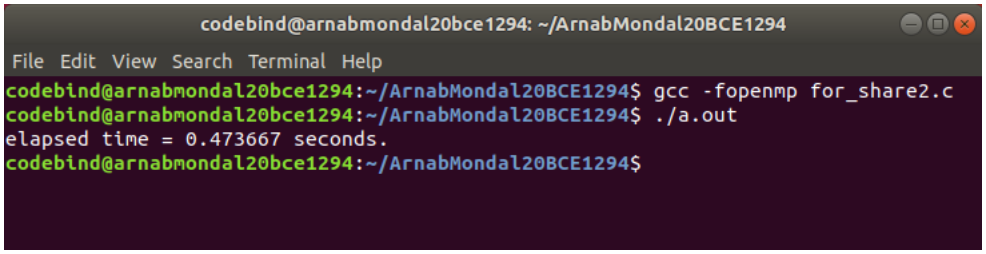

### Using nowait and dynamic

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>

#define N 500

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i, j, k;
    double elapsed, start, end;
    omp_set_num_threads(8);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            A[i][j] = 2;
            B[i][j] = 2;
        }

    start = omp_get_wtime();
//#pragma omp parallel for private(i,j,k) shared(A,B,C) schedule(dynamic)
ordered nowait
#pragma omp for private(i, j, k) schedule(dynamic) ordered nowait
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
        {
            for (k = 0; k < N; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
```

```
            }
        }
    }

    end = omp_get_wtime();

    elapsed = end - start;
    printf("elapsed time = %f seconds.\n", elapsed);
}
```

Output:



# Lab Submission – 04


Name : Abhishek N N
Reg.No : 20BCE1025
Email : abhishek.nn2020@vitstudent.ac.in


**Program: B.Tech**

**Semester: Fall 2022-23**

**Course: CSE4001 – Parallel and Distributed Computing**

**Date: 11-09-2022**

**Exercise: 04**

1. Perform calculator with four basic arithmetic operations that covers add, sub, multiply and modulo on each operation that is carried by different threads using sections.

Code:

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

int main()
{
    int n1= 2147483647, n2 = 2147483627;
    double st, en;
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                int res1 = n1 + n2;
                printf("Time : %f\n", (en - st));
                printf("Result : %d\n", res1);
```

```
            }
            #pragma omp section
            {
                int res2 = n1 - n2;
                printf("Time : %f\n", (en - st));
                printf("Result : %d\n", res2);
            }
            #pragma omp section
            {
                int res3 = n1 * n2;
                printf("Time : %f\n", (en - st));
                printf("Result : %d\n", res3);
            }
            #pragma omp section
            {
                int res4 = n1 % n2;
                printf("Time : %f\n", (en - st));
                printf("Result : %d\n", res4);
            }
        }
    }
    return 0;
}
```

Output:

```
codebind@arnabmondal20bce1294: ~/ArnabMondal20BCE1294/CSE4001_PDC
File  Edit  View  Search  Terminal  Help
codebind@arnabmondal20bce1294:~/ArnabMondal20BCE1294/CSE4001_PDC$ gcc -fopenmp l
ab4_1.c
codebind@arnabmondal20bce1294:~/ArnabMondal20BCE1294/CSE4001_PDC$ ./a.out
Time : 0.000000
Result : -22
Time : 0.000000
Result : 20
Time : 0.000001
Result : 21
Time : 0.000000
Result : 20
codebind@arnabmondal20bce1294:~/ArnabMondal20BCE1294/CSE4001_PDC$
```

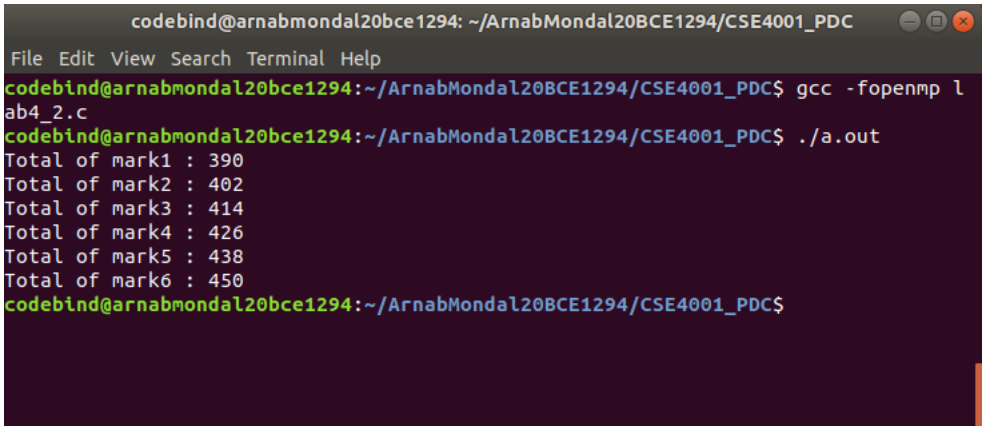
2. Find the class students' subject wise total marks in any 6 six subjects. utilize array and reduction ensure the variable clauses (for ex: private /shared).

Code:

```
#include<stdio.h>
#include<omp.h>
int main()
{
```

```
  int arr[6][6];
  int i,j;
  int sum = 0;
  for(i = 0;i < 6;i++)
  {
  sum += 10;
    for(j = 0;j < 6;j++)
    {
      //printf("Enter %d student's marks:",i + 1);
      //scanf("%d",&arr[i][j]);
      arr[i][j] = sum;
      sum += 2;
    }
  }
  int mark1, mark2, mark3, mark4, mark5, mark6;
  mark1=mark2=mark3=mark4=mark5=mark6=0;
  sum = 0;
  #pragma omp parallel for reduction(+ : mark1) reduction(+ : mark2)
reduction(+ : mark3) reduction(+ : mark4) reduction(+ : mark5) reduction(+ :
mark6) shared(arr) private(i) ordered
     for(i = 0;i < 6;i++)
     {
    mark1 += arr[i][0];
    mark2 += arr[i][1];
    mark3 += arr[i][2];
    mark4 += arr[i][3];
    mark5 += arr[i][4];
    mark6 += arr[i][5];
     }
     printf("Total of mark1 : %d\n",mark1);
     printf("Total of mark2 : %d\n",mark2);
     printf("Total of mark3 : %d\n",mark3);
     printf("Total of mark4 : %d\n",mark4);
     printf("Total of mark5 : %d\n",mark5);
     printf("Total of mark6 : %d\n",mark6);
 return 0;
}
```

Output:



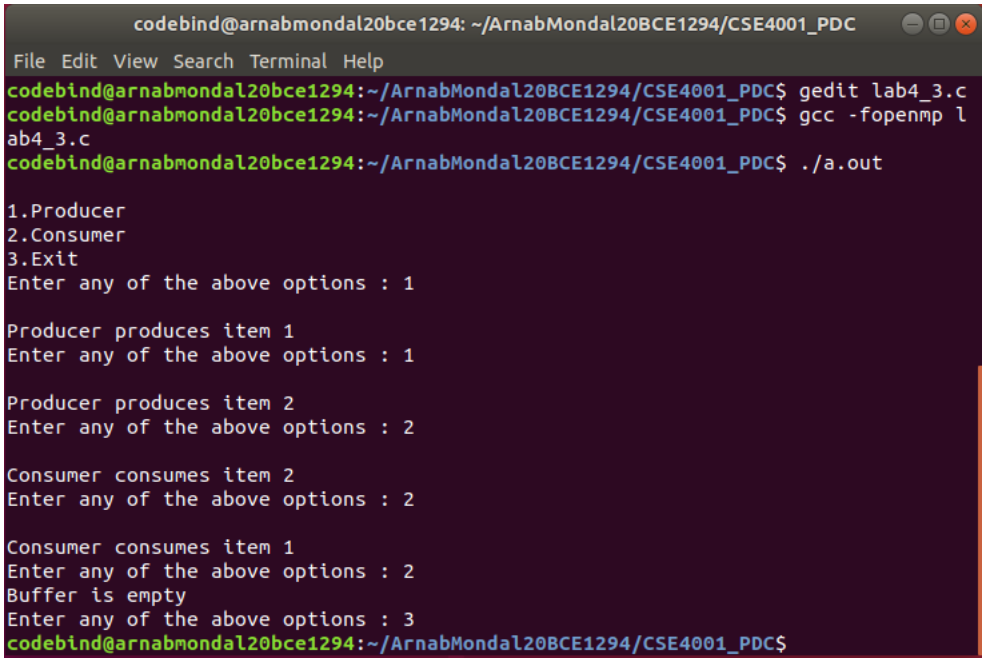

3. Write the producer consumer problem using critical section directives.

Code:

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

int mutex = 1;
int full = 0;

int empty = 10, x = 0;

void producer()
{
    --mutex;
    ++full;
    --empty;
    x++;
    printf("\nProducer produces item %d", x);
    ++mutex;
}

void consumer()
{
    --mutex;
    --full;
    ++empty;
    printf("\nConsumer consumes item %d", x);
    x--;
    ++mutex;
}

int main()
{
    int n, i;
```

```
    printf("\n1.Producer"
        "\n2.Consumer"
        "\n3.Exit");
#pragma omp critical

    for (i = 1; i > 0; i++) {

        //printf("\n1.Producer"
        //"\n2.Consumer"
        //"\n3.Exit");
        printf("\nEnter any of the above options : ");
        scanf("%d", &n);

        switch (n) {
        case 1:
            if ((mutex == 1)
                && (empty != 0)) {
                producer();
            }

            else {
                printf("Buffer is full");
            }
            break;

        case 2:

            if ((mutex == 1)
                && (full != 0)) {
                consumer();
            }

            else {
                printf("Buffer is empty");
            }
            break;

        case 3:
            exit(0);
            break;
        }
    }
}
```

Output:


codebind@arnabmondal20bce1294: ~/ArnabMondal20BCE1294/CSE4001_PDC
File Edit View Search Terminal Help
codebind@arnabmondal20bce1294:~/ArnabMondal20BCE1294/CSE4001_PDC$ gedit lab4_3.c
codebind@arnabmondal20bce1294:~/ArnabMondal20BCE1294/CSE4001_PDC$ gcc -fopenmp l
ab4_3.c
codebind@arnabmondal20bce1294:~/ArnabMondal20BCE1294/CSE4001_PDC$ ./a.out
1.Producer
2.Consumer
3.Exit
Enter any of the above options : 1
Producer produces item 1
Enter any of the above options : 1
Producer produces item 2
Enter any of the above options : 2
Consumer consumes item 2
Enter any of the above options : 2
Consumer consumes item 1
Enter any of the above options : 2
Buffer is empty
Enter any of the above options : 3
codebind@arnabmondal20bce1294:~/ArnabMondal20BCE1294/CSE4001_PDC$

# Lab Submission – 05


Name : Abhishek N N
Reg.No : 20BCE1025
Email : abhishek.nn2020@vitstudent.ac.in


**Program: B.Tech**

**Semester: Fall 2022-23**

**Course: CSE4001 – Parallel and Distributed Computing**

**Date: 07-09-2022**

**Exercise: 05**

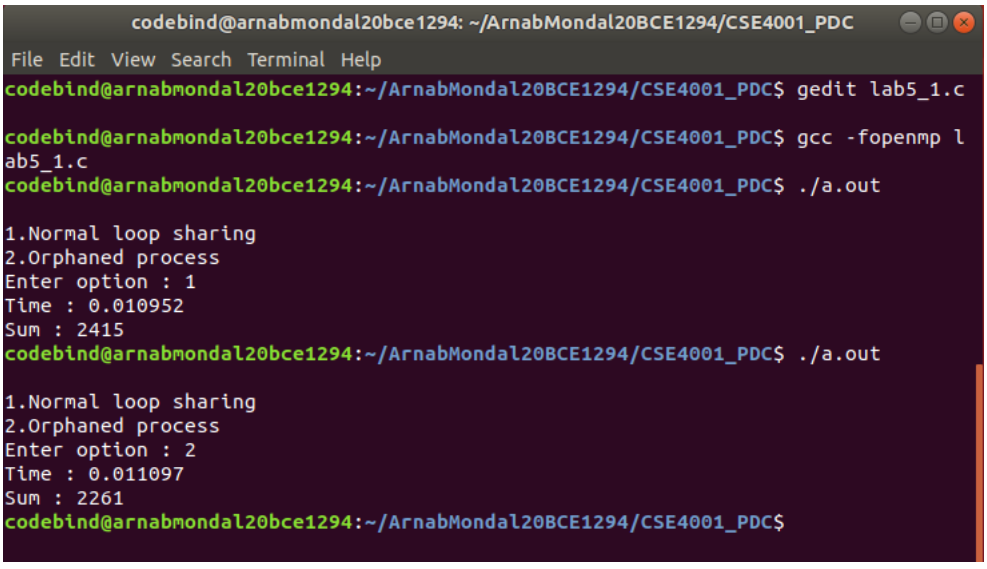
1. Do the Combined and Orphaned parallel loop reduction for marks total of one subject for 70 students.

   Code:

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

void cal(int arr[])
{
    int sum = 0;
    double st, en;
    st = omp_get_wtime();
    #pragma omp parallel for
    for(int i=0;i<70;i++)
        {
            sum += arr[i];
        }
    en = omp_get_wtime();
    printf("Time : %f\n", (en - st));
    printf("Sum : %d\n", sum);
}
```

```c
int main()
{
    int arr[70], opt;
    double st, en;
    printf("\n1.Normal loop sharing\n2.Orphaned process\nEnter option : ");
    scanf("%d", &opt);
    for(int i=0;i<70;i++)
    {
        arr[i] = i;
    }
    int sum = 0;
    if(opt == 1)
    {
        st = omp_get_wtime();
        #pragma omp parallel for
        for(int i=0;i<70;i++)
            {
                sum += arr[i];
            }
        en = omp_get_wtime();
        printf("Time : %f\n", (en - st));
        printf("Sum : %d\n", sum);
    }
    else if(opt == 2)
    {
        cal(arr);
    }
    return 0;
}
```

Output:

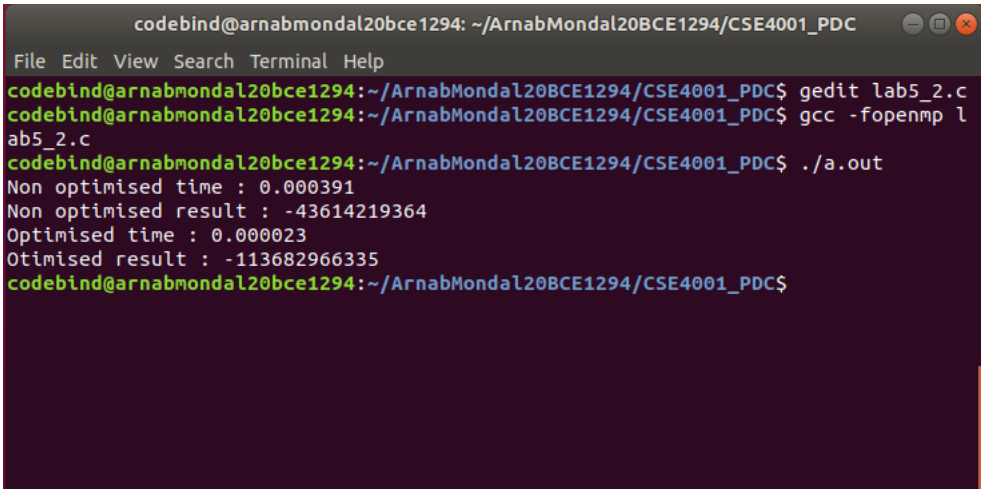



2. Show one optimization Technique

Code:

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

int main()
{
    int arr[100], opt;
    double st, en;
    for(int i=0;i<70;i++)
    {
        arr[i] = 50;
    }
    long res = 0;
    int x=100, y=2000, z=10750;
    int temp1 = (x*y*z) / 3;
    int temp2 = (x+y+z) / 7;
    temp1 += temp2;
    st = omp_get_wtime();
    #pragma omp parallel for
    for(int i=0;i<100;i++)
        {
            res += (((x*y*z) / 3) + ((x+y+z) / 7)) + i;
        }
    en = omp_get_wtime();
    printf("Non optimised time : %f\n", (en - st));
    printf("Non optimised result : %ld\n", res);
    st = omp_get_wtime();
    #pragma omp parallel for
    for(int i=0;i<100;i++)
        {
            res += temp1 + i;
        }
    en = omp_get_wtime();
    printf("Optimised time : %f\n", (en - st));
    printf("Otimised result : %ld\n", res);
    return 0;
}
```

Output:


codebind@arnabmondal20bce1294: ~/ArnabMondal20BCE1294/CSE4001_PDC
File Edit View Search Terminal Help
codebind@arnabmondal20bce1294:~/ArnabMondal20BCE1294/CSE4001_PDC$ gedit lab5_2.c
codebind@arnabmondal20bce1294:~/ArnabMondal20BCE1294/CSE4001_PDC$ gcc -fopenmp l
ab5_2.c
codebind@arnabmondal20bce1294:~/ArnabMondal20BCE1294/CSE4001_PDC$ ./a.out
Non optimised time : 0.000391
Non optimised result : -43614219364
Optimised time : 0.000023
Otimised result : -113682966335
codebind@arnabmondal20bce1294:~/ArnabMondal20BCE1294/CSE4001_PDC$

# Lab Submission – 09


Name : Abhishek N N
Reg.No : 20BCE1025
Email : abhishek.nn2020@vitstudent.ac.in


**Program: B.Tech**

**Semester: Fall 2022-23**

**Course: CSE4001 – Parallel and Distributed Computing**

**Date: 27-09-2022**

**Exercise: 09**

QUESTION:

Sum 10 elements using array ( use all 6 basic fucntions of MPI)
Using MPI_Send() and MPI_Recv() functions.

## Code:

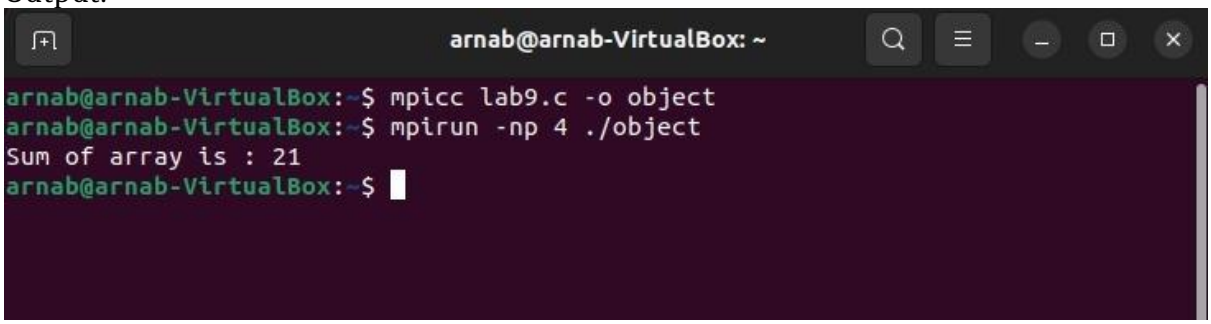
```
#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

// size of array
#define n 10

int a[] = {1,2,3,4,5,6};

// 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:




RESULT –SUM OF THE ARRAY IS FOUND TO BE EQUAL TO 21.

# Lab Submission – 10


Name : Abhishek N N
Reg.No : 20BCE1025
Email : abhishek.nn2020@vitstudent.ac.in


**Program: B.Tech**

**Semester: Fall 2022-23**

**Course: CSE4001 – Parallel and Distributed Computing**

**Date: 03-10-2022**

**Exercise: 10**

1. Use inbuilt MPI_Bcast and MPI_Ibcast for broadcasting the message from root to all the other process
2. Use ur own function broad_cast with MPI Send and Recv

OPTional for practice:
Use MPI_Wtime ,MPI_Barrier if required to check the time among MPI_Bcast, MPI_Ibcast and ur own function.

**bcast.c**

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>
int main(int argc, char **argv)
{
    char message[20];
    int i,rank,size;
    MPI_Status status;
    int root=0;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank==root)
    {
```

```
        strcpy(message, "Hello World");
    }
    MPI_Bcast(message,13,MPI_CHAR, root, MPI_COMM_WORLD);
    printf("Message from process %d: %s\n", rank, message);
    MPI_Finalize();
}
```

**Output:**

```
arnab@arnab-VirtualBox:~/Desktop$ mpirun -np 4 ./object
Message from process 0: Hello World
Message from process 2: Hello World
Message from process 1: Hello World
Message from process 3: Hello World
```

**ibcast.c**

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
int main(int argc, char **argv)
{
    char message[20];
    int i,rank,size;
    MPI_Status status;
    MPI_Request request = MPI_REQUEST_NULL;
    int root=0;MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank==root)
    {
        strcpy(message, "Hello World");
    }
    MPI_Ibcast(message,13,MPI_CHAR, root, MPI_COMM_WORLD, &request);
    MPI_Wait(&request, &status);
    if(rank==root)
    {
        strcpy(message, "What will happen?");
    }
    printf("Message from process %d: %s\n", rank, message);
    MPI_Finalize();
}
```

**Output:**

```
arnab@arnab-VirtualBox:~/Desktop$ gedit ibcast.c
arnab@arnab-VirtualBox:~/Desktop$ mpicc ibcast.c -o object
arnab@arnab-VirtualBox:~/Desktop$ mpirun -np 4 ./object
Message from process 3: Hello World
Message from process 0: What will happen?
Message from process 1: Hello World
Message from process 2: Hello World
```

**cast2.c**

```c
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
void my_bcast(void* data, int count, MPI_Datatype datatype, int root,MPI_Comm
communicator)
{
    int world_rank;
    MPI_Comm_rank(communicator, &world_rank);
    int world_size;
    MPI_Comm_size(communicator, &world_size);
    if(world_rank == root)
    {   // If we are the root process, send our data to everyone
        int i;
        for(i = 0; i < world_size; i++)
        {
            if(i != world_rank)
            {
                MPI_Send(data, count, datatype, i, 0, communicator);
            }
        }
    }
    else
    {   // If we are a receiver process, receive the data from the root
        MPI_Recv(data, count, datatype, root, 0, communicator,
MPI_STATUS_IGNORE);
    }
}
int main(int argc, char** argv)
{
    MPI_Init(NULL, NULL);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int data;
    if(world_rank == 0)
    {
        data = 100;
        printf("Process 0 broadcasting data %d\n", data);
```

```
        my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    }
    else
    {
        my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
        printf("Process %d received data %d from root process\n", world_rank,
data);
    }
    MPI_Finalize();
}
```

**Output:**

```
arnab@arnab-VirtualBox:~/Desktop$ gedit cas2.c
arnab@arnab-VirtualBox:~/Desktop$ mpicc cas2.c -o object
arnab@arnab-VirtualBox:~/Desktop$ mpirun -np 4 ./object
Process 0 broadcasting data 100
Process 2 received data 100 from root process
Process 3 received data 100 from root process
Process 1 received data 100 from root process
```

# Lab Submission – 11


Name : Abhishek N N
Reg.No : 20BCE1025
Email : abhishek.nn2020@vitstudent.ac.in


**Program: B.Tech**

**Semester: Fall 2022-23**

**Course: CSE4001 – Parallel and Distributed Computing**

**Date: 17-10-2022**

**Exercise: 11**

1. Use Gather to get array 0f [30] where p1 has a1[10], p2 has a2[10] and p3 has a3[10].
2. Use Reduce to compute sum of 10 elements

Gather:

**Code:**

```
#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <mpi.h>

//using namespace std;

int main(int argc, char** argv)

{
```

```
int i,j,k,p,a[30],b[10],c[10], d[10],myrank,res,x,y,interval,sum = 0;


for(i=0;i<10;i++)

{

    b[i] = i;

    c[i] = i;

    d[i] = i;

}


MPI_Status status;

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Comm_size(MPI_COMM_WORLD, &p);


MPI_Gather(&b,10,MPI_INT, a, 10,MPI_INT,0,MPI_COMM_WORLD);

MPI_Gather(&c,10,MPI_INT, a, 10,MPI_INT,0,MPI_COMM_WORLD);

MPI_Gather(&d,10,MPI_INT, a, 10,MPI_INT,0,MPI_COMM_WORLD);


if(myrank == 0)

{

    for(i=0;i<30;i++)

        printf("%d ",a[i]);

}


MPI_Finalize();
```

```
    return 0;

}
```

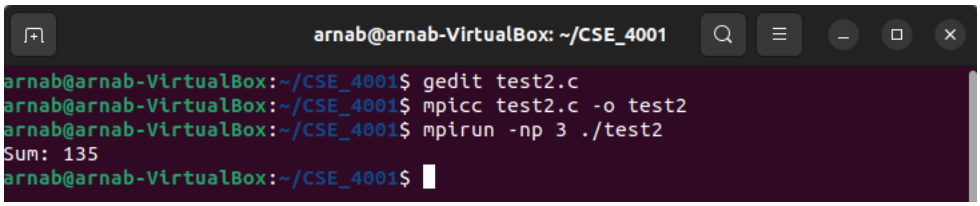
**Output:**


**Code:**

```
#include<stdio.h>

#include<stdlib.h>

#include<mpi.h>

#include<unistd.h>

int main(int argc,char **argv){

  int np,myrank;

  int sum = 0;

  for(int i = 0;i < 10;i++){

    sum = sum + i;

  }

  MPI_Status status;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

  MPI_Comm_size(MPI_COMM_WORLD,&np);

  int final_sum = 0;
```

```c
MPI_Reduce(&sum,&final_sum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

if(myrank == 0){

  printf("Sum: %d\n",final_sum);

}

MPI_Finalize();

return 0;

}
```

**Output:**



# Lab Submission – 11


Name : Abhishek N N
Reg.No : 20BCE1025
Email : abhishek.nn2020@vitstudent.ac.in


**Program: B.Tech**

**Semester: Fall 2022-23**

**Course: CSE4001 – Parallel and Distributed Computing**

**Date: 31-10-2022**

**Exercise: 11**

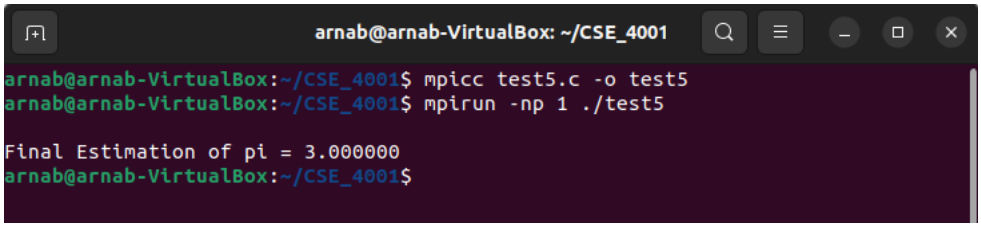
1. Estimating PI using Monte Carlo Method
2. Calculate sum of N Prime Numbers

**Code1:**

```c
#include <mpi.h>

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <time.h>

#define INTERVAL 10000

int main()

{

    int interval, i;
```

```c
    double rand_x, rand_y, origin_dist, pi;

    int circle_points = 0, square_points = 0;

    srand(time(NULL));

    for (i = 0; i < (INTERVAL * INTERVAL); i++)
    {

        rand_x = (double)(rand() % (INTERVAL + 1)) / INTERVAL;

        rand_y = (double)(rand() % (INTERVAL + 1)) / INTERVAL;

        origin_dist = rand_x * rand_x + rand_y * rand_y;

        if (origin_dist <= 1)

            circle_points++;

        square_points++;

        pi = (4 * circle_points) / square_points;
    }

    printf("\nFinal Estimation of pi = %f\n", pi);

    return 0;
}
```

**Output1:**




**Code 2:**

```c
#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <unistd.h>
```

```
int check_prime(int a)

{

    int c;

    for (c = 2; c <= a - 1; c++)

    {

        if (a % c == 0)

            return 0;
    }

    return 1;
}

int main(int argc, char **argv)
{

    int np, myrank;

    int limit = 5;

    int sum = 0;

    for (int i = 2; i <= limit; i++)
    {

        if (check_prime(i) == 1)

        {

            sum += i;
        }
    }

    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    MPI_Comm_size(MPI_COMM_WORLD, &np);

    int final_sum = 0;
```

```c
    MPI_Reduce(&sum, &final_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myrank == 0)
    {

        printf("Sum: %d\n", final_sum);
    }

    MPI_Finalize();

    return 0;
}
```

**Output2:**