HW2

Que1.1) C code attached as hw2_a1.c

```
#include "mpi.h"
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>


#define pi 3.14159265358979323846  /* pi */

/*MPI_Type Enum*/
enum {
	MASTER = 0,
	FROM_MASTER = 1,
	FROM_WORKER = 2,
};

double estimate_g(double lower_bound, double upper_bound, long long int N)
{
	int			num_tasks;		/* number of tasks     */
	int			task_id;	/* number of processes  */
	int			num_workers;		/* number of worker tasks */
	int			source;			/* rank of sender       */
	int			dest;			/* rank of receiver     */
	int			mtype;			/* message type */
	MPI_Status		status;			/* return status for receive */
	int			rows;
	double			x, y;			/* First boundary condition, Second boundary
condition */
	double			sum1 = 0, sum = 0;
	int			i;
	clock_t begin, end;
	double time_spent;

	/* Find out process rank  */
	MPI_Comm_rank(MPI_COMM_WORLD, &task_id);

	/* Find out number of processes */
	MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);



	num_workers = num_tasks - 1;
	int average_row = N / num_workers;
	int left_overs = N % num_workers;
```

```
	if (task_id == MASTER) {
		//begin = clock();
		//Send matrix data to worker tasks
		mtype = FROM_MASTER;
		for (dest = 1; dest < num_tasks; dest++) {
			rows = (dest <= left_overs) ? average_row + 1 : average_row;
			printf("Sending %d rows to task %d\n", rows, dest);
			MPI_Send(&lower_bound, 1, MPI_DOUBLE, dest, mtype,
MPI_COMM_WORLD);
			MPI_Send(&upper_bound, 1, MPI_DOUBLE, dest, mtype,
MPI_COMM_WORLD);
			MPI_Send(&N, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
			MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
		}
		printf("Sent all the data!\n");

	}
	else {
		//Worker task
		mtype = FROM_MASTER;

		MPI_Recv(&lower_bound, 1, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD,
&status);
		MPI_Recv(&upper_bound, 1, MPI_DOUBLE, MASTER, mtype,
MPI_COMM_WORLD, &status);
		MPI_Recv(&N, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
		MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
		double result, final_val = 0, sum = 0;
		int j;
		srand(time(NULL)*task_id);
		for (j = 0; j < rows; j++)
		{
			result = exp(-1 * pow(2 * ((rand() / (RAND_MAX / (upper_bound -
lower_bound))) + lower_bound), 2));
			final_val += result;
		}
		sum1 = (8 * sqrt(2 * pi))*((upper_bound - lower_bound) / N)*final_val;
	}
	return sum1;
}

void collect_results(double *result)
{
	int			num_tasks;		/* number of tasks     */
	int			task_id;	/* number of processes  */
	int			source;			/* rank of sender      */
	int			dest;			/* rank of receiver     */
```

```
	int			mtype;			/* message type */
	MPI_Status		status;			/* return status for receive */
	double			sum1 = 0, sum = 0;
	int			i;


	/* Find out number of processes */
	MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);

	/* Find out process rank  */
	MPI_Comm_rank(MPI_COMM_WORLD, &task_id);

	sum1 = *result;

	//Wait for results from workers
	if (task_id == MASTER)
	{
		mtype = FROM_WORKER;
		for (i = 1; i < num_tasks; i++) {
			source = i;
			printf("Receiving data from task %d\n", source);
			MPI_Recv(&sum1, 1, MPI_DOUBLE, source, mtype,
MPI_COMM_WORLD, &status);
			sum += sum1;
		}
		//Print the results
		printf("\n\nValue of integration is:%lf\n", sum);
	}else
	{
		//Send the results back to the Master process
		mtype = FROM_WORKER;
		MPI_Send(&sum1, 1, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
	}
}

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

	int			num_tasks;		/* number of tasks     */
	int			task_id;	/* number of processes  */
	clock_t begin, end;
	double time_spent;



	//MPI
	/* Start up MPI */
	MPI_Init(&argc, &argv);
```

```
	/* Find out number of processes */
	MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);

	/* Find out process rank  */
	MPI_Comm_rank(MPI_COMM_WORLD, &task_id);

	float lower_bound = atof(argv[1]);
	float upper_bound = atof(argv[2]);
	long long int N = atof(argv[3]);

	begin = clock();

	double result = estimate_g(lower_bound, upper_bound, N);
	collect_results(&result);

	MPI_Finalize();
	if (task_id == MASTER) {

		end = clock();
		time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
		printf("%d  number of processors and time spent %f sec\n", num_tasks,
time_spent);
	}
	return 0;
}
```

Que1.2) C code attached as hw2_a2.c

```
#include "mpi.h"
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>


#define pi 3.14159265358979323846  /* pi */

/*MPI_Type Enum*/
enum {
	MASTER = 0,
	FROM_MASTER = 1,
	FROM_WORKER = 2,
};

double estimate_g(double lower_bound, double upper_bound, long long int N)
{
	int			num_tasks;		/* number of tasks     */
	int			task_id;	/* number of processes  */
	int			num_workers;		/* number of worker tasks */
```

```
	int			source;			/* rank of sender	*/
	int			dest;			/* rank of receiver	*/
	int			mtype;			/* message type */
	MPI_Status		status;			/* return status for receive */
	int			rows;
	double			x, y;			/* First boundary condition, Second boundary 
condition */
	double			sum1 = 0, sum = 0;
	int			i;
	clock_t begin, end;
	double time_spent;

	/* Find out process rank  */
	MPI_Comm_rank(MPI_COMM_WORLD, &task_id);

	/* Find out number of processes */
	MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);



	num_workers = num_tasks - 1;
	int average_row = N / num_workers;
	int left_overs = N % num_workers;

	if (task_id == MASTER) {
		//begin = clock();
		//Send matrix data to worker tasks
		mtype = FROM_MASTER;
		for (dest = 1; dest < num_tasks; dest++) {
			rows = (dest <= left_overs) ? average_row + 1 : average_row;
			printf("Sending %d rows to task %d\n", rows, dest);
			MPI_Send(&lower_bound, 1, MPI_DOUBLE, dest, mtype, 
MPI_COMM_WORLD);
			MPI_Send(&upper_bound, 1, MPI_DOUBLE, dest, mtype, 
MPI_COMM_WORLD);
			MPI_Send(&N, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
			MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
		}
		printf("Sent all the data!\n");

	}
	else {
		//Worker task
		mtype = FROM_MASTER;

		MPI_Recv(&lower_bound, 1, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, 
&status);
```

```
			MPI_Recv(&upper_bound, 1, MPI_DOUBLE, MASTER, mtype,
MPI_COMM_WORLD, &status);
			MPI_Recv(&N, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
			MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
			double result, final_val = 0, sum = 0;
			int j;
			srand(time(NULL)*task_id);
			for (j = 0; j < rows; j++)
			{
				result = exp(-1 * pow(2 * ((rand() / (RAND_MAX / (upper_bound -
lower_bound))) + lower_bound), 2));
				final_val += result;
			}
			sum1 = (8 * sqrt(2 * pi))*((upper_bound - lower_bound) / N)*final_val;
		}
		return sum1;
}

void collect_results(double *result)
{
		int			num_tasks;		/* number of tasks	*/
		int			task_id;	/* number of processes	*/

		double			sum1 = 0, sum = 0;
		int			i;

		/* Find out number of processes */
		MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);

		/* Find out process rank  */
		MPI_Comm_rank(MPI_COMM_WORLD, &task_id);

		sum1 = *result;

		//Wait for results from workers
		MPI_Reduce(&sum1, &sum, num_tasks, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
		if (task_id == MASTER)
		{
			printf("\n\n HERE Value of integration is:%lf\n", sum);
		}
}

int main(int argc, char *argv[]) {
		int			num_tasks;		/* number of tasks	*/
		int			task_id;	/* number of processes	*/
		clock_t begin, end;
		double time_spent;
```

```
	//MPI
	/* Start up MPI */
	MPI_Init(&argc, &argv);

	/* Find out number of processes */
	MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);

	/* Find out process rank  */
	MPI_Comm_rank(MPI_COMM_WORLD, &task_id);

	float lower_bound = atof(argv[1]);
	float upper_bound = atof(argv[2]);
	long long int N = atof(argv[3]);

	begin = clock();

	double result = estimate_g(lower_bound, upper_bound, N);
	collect_results(&result);

	MPI_Finalize();

	if (task_id == MASTER) {

		end = clock();
		time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
		printf("time spent %f sec\n",  time_spent);
	}
	return 0;
}
```

Que1.3)

Ans:

Code 1.1) Tests, Results and Timing information:

Lower bound is 0 and upper bound is 0.06

Case 1: For processor cores 32 and samples 10000000
Value of integration is: 1.197431
32 number of processors and time spent 0.040000 sec.

Case 2: For processor cores 8 and samples 1000
Value of integration is: 1.197294
8 number of processors and time spent 0.000000 sec

Case 3: For processor cores 4 and samples 10000000

```
Value of integration is: 1.197434
4 number of processors and time spent 0.250000 sec

Case 4: For processor cores 4 and samples 100
Value of integration is: 1.197434
4 number of processors and time spent 0.000000 sec
```

```
Code 1.2) Tests, Results and Timing information:

Lower bound is 0 and upper bound is 0.06

Case 1: For processor cores 32 and samples 10000000
Value of integration is: 1.197432
32 number of processors and time spent 0.020000 sec.

Case 2: For processor cores 8 and samples 1000
HERE Value of integration is: 1.197336
8 number of processors and time spent 0.000000 sec

Case 3: For processor cores 4 and samples 10000000
Value of integration is: 1.197431
4 number of processors and time spent 0.200000 sec

Case 4: For processor cores 4 and samples 100
Value of integration is: 1.197394
4 number of processors and time spent 0.000000 sec
```

As calculation for million sample on 32 processor cores took far less time than same number of calculations on 4 processor cores as evident with case 1 and 3 in both codes results.

The Code was embarrassingly parallel as all parts of the code could be parallelized without much effort as there was no interdependency between any tasks or results.

Que2.1) C code attached as hw2_a2.c

```
/**********Source Code ***/
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include<time.h>
// CPU Calculation to check the output
#define MAX_RANGE 10001
#define MASTER 0
```

```
void out_results(int *tst, int Nw, int Nh)
{
	int i, j;
	for (i = 0; i < Nh; i++)
	{
		for (j = 0; j < Nw; j++)
			printf("%d ", tst[i*Nw + j]);
		printf("\n");
	}
}

void initialize_data(int **A, int **Ap,int N)
{
	int i, j, p,rank;
	srand(time(NULL));
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	/*Master node will have full memory allocation*/
	//if (rank == 0)
	{
		*A = (int *)malloc(N * N * sizeof(int));
		*Ap = (int *)calloc(N * N ,sizeof(int));
	}

	for (i = 0; i < N ; i++)
		for (j = 0; j < N ; j++)
			(*A)[i + N*j] =  rand()% MAX_RANGE;
/*	if (rank == 0)
	{
		printf("input matrix \n");
		out_results(*A,N,N);
	}
	*/
}
void scatter_data(int *A,int N)
{
	int i, p, rank,compute_size;
	int *offsets, *chunk_sizes;
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	offsets = (int *)malloc(p*sizeof(int)); /*offsets from which data is sent of input A from
master*/
	chunk_sizes = (int *)malloc(p*sizeof(int)); /*sizes of the data chunks sent of input A from
master*/

	compute_size = ((N - 2) / p);
	for (i = 0; i<p; ++i) {
```

```
			/*data chunks that would be computed at each node*/
			chunk_sizes[i] = (compute_size + 2)*N + (i==0)*((N-2)%p)*N;
			offsets[i] =  i*compute_size*N+(i!=0)*((N-2)%p)*N;
		}
		if (((N - 2) / p) > 0)
		/*Scatter data from the master to the worker*/
		MPI_Scatterv(A, chunk_sizes, offsets, MPI_INT, A, (compute_size+2)*N, MPI_INT,
			MASTER, MPI_COMM_WORLD);

		free(chunk_sizes);
		free(offsets);
}
void mask_operation(int *A, int N, int * Ap)
{
		int i, j, p, rank, compute_size;
		int *res ;
		MPI_Comm_size(MPI_COMM_WORLD, &p);
		MPI_Comm_rank(MPI_COMM_WORLD, &rank);

		/*Values converted to float before computation to avoid accumulation from overflowing as
MAX_RANGE 10000 */
		/*Offset added to avoid first row overwrite for master*/
		if (rank == 0)
		{
			compute_size = ((N - 2) / p)+((N-2)%p);/*data chunks + leftovers that would be
computed at each node*/
			res = Ap +N;
		}
		else
		{
			compute_size = ((N - 2) / p);/*data chunks that would be computed at each node*/
			res = Ap;
		}
		for (i = 1; i < compute_size+1; i++)
		{
			for (j = 1; j < N - 1; j++)
			{
				res[(i-1)*N + j] =(int)( ((float) A[(i - 1) * N + j - 1] + (float)A[(i - 1)*N + j] +
(float)A[(i - 1)*N + j + 1] + (float)A[i *N + j - 1] +
					2 * ((float)A[i*N + j]) + (float)A[i*N + j + 1] + (float)A[(i + 1)*N
+ j - 1] + (float)A[(i + 1)*N + j] + (float)A[(i + 1)*N + j + 1]) / 10);
			}
		}

}
void gather_results(int *Ap, int N) {
		int i, p, rank, compute_size;
		int *offsets, *chunk_sizes;
```

```
		int *res;
		MPI_Comm_size(MPI_COMM_WORLD, &p);
		MPI_Comm_rank(MPI_COMM_WORLD, &rank);
		offsets = (int *)malloc(p*sizeof(int));/*offsets from which data is sent of input A from
master*/
		chunk_sizes = (int *)malloc(p*sizeof(int));
		compute_size = ((N - 2) / p);
		for (i = 0; i<p; ++i) {
			 /*data chunks that would be computed at each node*/
			chunk_sizes[i] = (compute_size + 2)*N + (i==0)*((N-2)%p)*N;
			offsets[i] =  i*compute_size*N+(i!=0)*((N-2)%p)*N+N;
		}
		/*Offset added to avoid first row overwrite for master*/
		if (rank == 0)
			res = Ap +N;
		else
			res = Ap;
		// Gather the geenerated outputs
		if (((N - 2) / p) > 0)
		MPI_Gatherv(res, compute_size*N, MPI_INT, Ap, chunk_sizes, offsets, MPI_INT,
			MASTER, MPI_COMM_WORLD);
/*		if (rank == 0)
		{
			printf("Output Matrix\n");
			out_results(Ap, N,N);
		}*/
		free(chunk_sizes);
		free(offsets);
}
int main(int argc, char** argv) {

		int * A, *Ap,*tst;
		int p, rank;
		clock_t begin,end;
		double time_spent;
		MPI_Init(&argc, &argv);
		int N = atoi(argv[1]);
		begin = clock();
		MPI_Comm_size(MPI_COMM_WORLD, &p);
		MPI_Comm_rank(MPI_COMM_WORLD, &rank);
		initialize_data(&A, &Ap,N);
		scatter_data(A, N);
		mask_operation(A, N, Ap);
		gather_results(Ap, N);
		if (rank == MASTER) {

			end = clock();
			time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
```

```
        printf("time spent %f sec\n",  time_spent);
    }
    MPI_Finalize();

    free(A);
    free(Ap);
    return 0;
}
```

Que2.2)
Performance Results with fixed Matrix size with varying Number of Cores

```
Case 1: 4 cores, N 10000 time spent 2.490000 sec
Case 2: 8 cores, N 10000 time spent 1.560000 sec
Case 2: 16 cores, N 10000 time spent 0.930000 sec
Case 2: 32 cores, N 10000 time spent 0.540000 sec
```

The scaling of performance with the number of cores is not a linear function. It always actually less than the ratio of the number of the cores, and depends on the workload on all the cores as well as whether the cores are on the same node or not.

Performance Results with varying Matrix size and fixed Number of Core

```
Case 2: 8 cores, N 100   time spent 0.000-00 sec
Case 3: 8 cores, N 500   time spent 0.010000 sec
Case 3: 8 cores, N 1000  time spent 0.020000 sec
Case 4: 8 cores, N 5000  time spent 0.370000 sec
Case 4: 8 cores, N 10000 time spent 1.560000 sec
```

With the increase in the number of elements in a matrix the performance decreases and is slightly higher drop than the ratio of the number of elements in matrix. Like from N= 5000 to N = 10000 the performance decrease is slightly more than 4 times.